IAN SINCLAIR

ORIC AND ATMOS MACHINE CODE

Oric and Atmos Machine Code

Oric and Atmos Machine Code

Ian Sinclair

GRANADA
London Toronto Sydney New York

Granada Technical Books
Granada Publishing Ltd
8 Grafton Street, London W1X 3LA

First published in Great Britain by
Granada Publishing 1984

British Library Cataloguing in Publication Data
Sinclair, Ian R.
Oric and Atmos machine code.
1. Oric computers——Programming
2. Machine codes (Electronic computers)
I. Title
007.64'25 QA76.8.07
ISBN 0-246-12150-5

Typeset by V & M Graphics Ltd, Aylesbury, Bucks
Printed and bound in Great Britain by
Mackays of Chatham, Kent

Contents

Preface	vii
1 ROM, RAM, Bytes and Bits	1
2 Digging inside the Oric/Atmos	14
3 The Microprocessor	28
4 6502 Details	40
5 Register Actions	53
6 Taking a Bigger Byte	66
7 Ins and Outs and Roundabouts	86
8 Debugging, Checking and Oricmon	101
9 Last Round-up	113
Appendix A: How Numbers are Stored	129
Appendix B: Hex and Denary Conversions	131
Appendix C: The Instruction Set	133
Appendix D: Addressing Methods of the 6502	139
Appendix E: Passing Parameters	141
Appendix F: A CTRL Z Reset Action	144
Index	146

Preface

Many computer users are content to program in BASIC for all of their computing lives. A large number of others are eager to find out more about computing and their computer than the use of BASIC can lead to. Few, however, make much progress towards the use of machine code, which allows so much more control over the computer. The reason for this, to my mind, is that so many books which deal with machine code programming seem to start with the assumption that the reader is already familiar with the ideas and the words of this type of programming. Furthermore, many of these books treat machine code programming as a study in itself, leaving the reader with little clue as to how to apply machine code to his or her computer. An additional problem is that so many books on machine code programming emphasise mathematical programming, despite the fact that this is the least likely application that the user will consider.

This book has two main aims. One is to introduce Oric-1 and Oric Atmos owners to some of the details of how their microcomputers work, so leading to more effective programming even without delving into machine code. The second aim is to introduce the methods of machine code programming in a simple way. I must emphasise the word 'introduce'. No single book can tell you all about machine code, even the machine code for one computer. All I can claim is that I can get you started. Getting started means that you will be able to write short machine code routines, understand machine code routines that you see printed in magazines, and generally make more effective use of your Oric-1 or Oric Atmos. It also means that you will be able to make use of the many more advanced books on the subject, and progress to greater mastery of this fascinating topic.

Throughout the book, you will see references to the 'Oric/Atmos'. The machine code that is used is identical for both computers, and you will have to learn the same techniques whether you use the Oric-1 or the newer Atmos. The programs in this book have all been written for and tested on a 48K Atmos, however. In some examples, changes to address numbers will be needed to allow these programs to run on the Oric-1, but these are pointed out in the text. In Chapter 8, the version of Oricmon that has been used as an example was the version for the Oric-1. Though this would not run on the Atmos, the commands are identical to those that will be used on the Atmos version when it is available. Atmos owners should note carefully that many items of machine code software which are advertised for the Oric-1 will not run correctly on the Atmos without changes.

Understanding the operating system of your Oric/Atmos, and having the ability to work in machine code can open up an entirely new world of computing to you. This is why you find that most of the really spectacular games are written in machine code, and you will also find that many programs which are written mainly in BASIC will incorporate pieces of machine code in order to make use of its greater speed and better control of the computer.

I am very grateful to several people who made the production of this book possible. Of these, Richard Miles of Granada Publishing commissioned the book, and obtained an Atmos 48K for me. He, Sue Moore and Prue Harrison then did their usual miracles of meticulous effort on my manuscript. I am most grateful to Oric International for the loan of a 48K Atmos, a machine which I greatly liked. I sincerely hope that this book will open up the horizons of many Oric/Atmos owners to what their machines can do once they have broken out of the restraints of BASIC.

Ian Sinclair

Chapter One
ROM, RAM, Bytes and Bits

One of the things that discourage computer users from attempting to go beyond BASIC is the number of new words that spring up. The writers of many computing books, especially machine code computing, seem to assume that the reader has an electronics background and will understand all of the terms. I shall assume that you have no such background. All I shall assume is that you possess an Oric-1 or Atmos, and that you have some experience in programming your Oric or Atmos in BASIC. This means that we start at the correct place, which is the beginning. I don't want in this book to have to interrupt important explanations with technical or mathematical details, and these will be found in the Appendices. This way, you can read the full explanation of some points if you feel inclined. or skip them if you do not.

To start with, we have to think about *memory*. A unit of memory for a computer is, as far as we are concerned, just an electrical circuit that acts like a switch. You walk into a room, switch a light on, and you never think that it's remarkable in any way that the light stays on until you switch it off. You don't go about telling your friends that the light circuit contains a memory – and yet each memory unit of a computer is just a kind of miniature switch that can be turned on or off. What makes it a memory is that it will stay the way it has been turned, on or off, until it is changed. One unit of computer memory like this is called a *bit* – the name being short for *b*inary digi*t*. meaning a unit that can be switched to either one of two possible ways.

We'll stick with the idea of a switch, because it's very useful. Suppose that we wanted to signal with electrical circuits and switches. We could use a circuit like the one in Fig. 1.1. When the switch is on, the light is on, and we might take this as meaning YES. When the switch is turned off, the light goes out, and we might take this as meaning NO. You could attach any two meanings that you

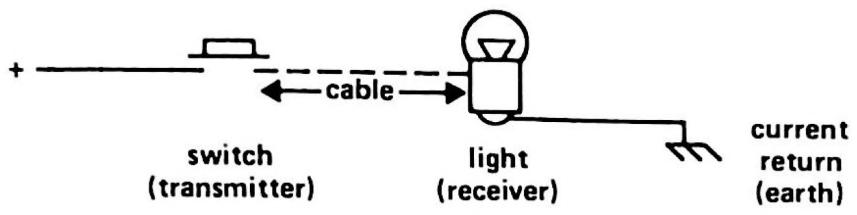
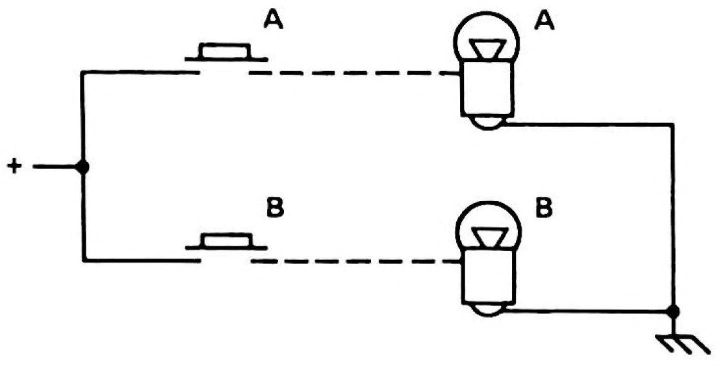


Fig. 1.1. A single-line switch and bulb signalling system.

liked to these two conditions (called 'states') of the light, so long as there are only two. Things improve if you can use two switches and two lights, as in Fig. 1.2. Now four different combinations are possible: (a) both off, (b) A on, B off, (c) A off, B on, (d) both on.



A	B
off	off
off	on
on	off
on	on

Fig. 1.2. Two-line signalling. Four possible signals can be sent.

This set of four possibilities means that we could signal four different meanings. Using one line allows two possible codes; using two lines allows four codes. If you feel inclined to work them all out, you'll find that using three lines will allow eight different codes. A moment's thought suggests that since 4 is 2X2, and eight is 2X2X2, then four lines might allow 2X2X2X2, which is 16, codes. It's true, and since we usually write 2X2X2X2 as 2^4 (two to the power 4), we can find out how many codes could be transmitted by any number of lines. We would expect eight lines, for example, to be able to carry 2^8 codes, which is 256. A set of eight switches, then, could be arranged so as to convey 256 different meanings. It's up to us to decide how we might want to use these signals.

One particularly useful way is called *binary code*. Binary code is a way of writing numbers using only two digits, ∅ and 1. We can think of Ø as meaning 'switch off' and 1 as meaning 'switch on'. 256 different numbers could be signalled, using eight switches, by thinking of Ø as meaning off and 1 as meaning on. This group of eight is called a *byte*, and it's the quantity that we use to specify the memory size of our computers. This is why the numbers 8 and 256 occur so much in machine code computing.

The way that the individual bits in a byte are arranged so as to indicate a number follows the same method that we use to indicate a number normally. When you write a number such as 256, the 6 means six units, the 5 is written to the immediate left of the 6 and means five tens, and the 2 is written one more place to the left and means two hundreds. These positions indicate the importance or significance of a digit, as Fig. 1.3 shows. The 6 in 256 is called the 'least significant digit', and the 2 is the 'most significant digit'. Change the 6 to 7 or 5, and the change is just one part in 256. Change the 2 to 1 or 3 and the change is one hundred parts in 256 – much more important.

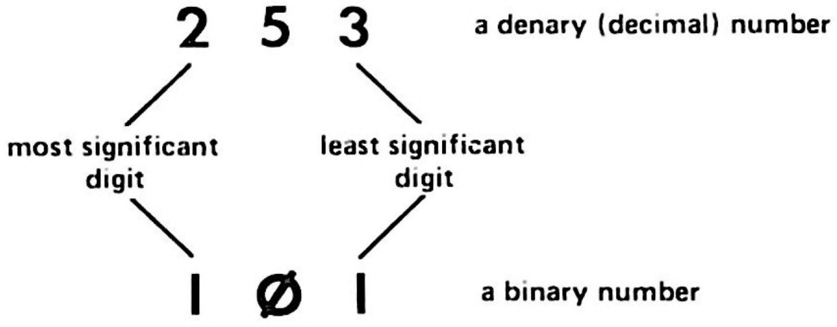


Fig. 1.3. The significance of digits. Our numbering system uses the position of a digit in a number to indicate its significance or importance.

Having looked at bits and bytes, it's time to go back to the idea of memory as a set of switches. As it happens, we need two types of memory in a computer. One type must be permanent, like mechanical switches or fixed connections, because it has to be used for retaining the number-coded instructions that operate the computer. This is the type of memory that is called ROM, meaning read-only memory. This implies that you can find out and copy what is in the memory, but not delete it or change it. The ROM is the most important part of your computer, because it contains all the instructions that make the computer carry out the actions of BASIC.

When you write a program for yourself, the computer stores it in the form of another set of number-coded instructions in a part of memory that can be used over and over again. This is a different type of memory that can be 'written' as well as 'read', and if we were logical about it we would refer to it as RWM, meaning read-write memory. Unfortunately, we're not very logical, and we call it RAM (meaning random-access memory). This was a name that was used in the very early days of computing to distinguish this type of memory from one which operated in a different way. We're stuck with the name of RAM now and probably forever!

The number code caper

Now we can get back to the bytes. We saw earlier that a byte, which is a group of eight bits, can consist of any one of 256 different arrangements of these bits. The most useful arrangement, however, is one that we call binary code. These different arrangements of binary code represent numbers which we write in ordinary form as ∅ to 255 (*not* 1 to 256, because we need a code for zero). Each byte of the 37631 bytes of RAM that are available in the 48K Oric/Atmos can store a number which is in this range of ∅ to 255.

Numbers by themselves are not of much use, and we wouldn't find a computer particularly useful if it could deal only with numbers between ∅ and 255, so we make use of these numbers as codes. Each number code can, in fact, be used to mean several different things. If you have worked with ASCII codes in BASIC, you will know that each letter of the alphabet and each of the digits ∅ to 9, and each punctuation mark, is coded in ASCII as a number between 32 (the space) and 127 (the left-arrow). That selection leaves you with a large number of ASCII code numbers which can be used for other purposes such as graphics characters. The ASCII code is not the only one, however. The Oric/Atmos uses its own coded meanings for numbers in this range of ∅ to 255. For example, when you type the word PRINT in a program line, what is placed in the memory of the Oric/Atmos (when you press ENTER) is not the sequence of ASCII codes for PRINT. This would be 80,82,73,78,84 – one byte for each letter. What is put into memory, in fact, is one byte, the binary form of the number 186. This single byte is called a 'token' and it can be used by the computer in two ways. One way is to locate the ASCII codes for the characters that make up the word PRINT. These are stored in the ROM, so that when you LIST a program you will see the word PRINT appear, not a character whose code is 186. The other, even more important, use of the 'token' is to locate a set of instructions which are also held in the ROM in the form of number codes. These instructions will cause characters to be printed on the screen, and the numbers that make up these codes are what we call 'machine code'. They control directly what the 'machine' does. That direct control is our reason for wanting to use machine code. When we use BASIC, the only commands we can use are the ones for which 'tokens' are provided. By using machine code, we can make up our own commands and do what we please. Incidentally, the fact that PRINT generates one 'token' is the reason why it is possible to use ? in place of PRINT. The Oric/Atmos has been designed so that a ? which is not placed between quotes will also cause 186 to be put into memory.

Do-it-yourself spot

As an aid to digesting all that information, try a short program. This one, in Fig. 1.4, is designed to reveal the keywords that are stored in the ROM, and it makes use of the BASIC instruction word PEEK.

```
10 FORN=49386TO49835
20 L=PEEK(N)
30 IF L>128THENPRINTCHR$(L-128)
40 IF L<128THENPRINTCHR$(L);
50 NEXT
```

Fig. 1.4. A program that reveals the keywords of Oric/Atmos BASIC.

PEEK has to be followed by a number or number variable within brackets, and it means 'find what byte is stored at this address number'. All of the bytes of memory within your Oric/Atmos are numbered from zero upwards, one number for each byte. Because this is so much like the numbering of houses in a road, we refer to these numbers as addresses. The action of PEEK is to find what number, which must be between ∅ and 255, is stored at each address. The Oric/Atmos automatically converts these numbers from the binary form in which they are stored into the ordinary decimal (more correctly, denary) numbers that we normally use. By using CHR$ in our program, we can print the character whose ASCII code is the number we have PEEKed at. The program uses the variable N as an address number, and then checks that PEEK(N) gives a number less than 128 – in other words a number which is an ASCII code. If it is, then the character is printed in line 40.

Now the reason that we have to check is that the last character in each set of words, or word, is coded in a different way. The number that we find for the last character has had 128 added to the ASCII code. For example, the first three address locations that the program PEEKs at contain the numbers 69, 78 and 196. The number 69 is the ASCII code for E, 78 is the code for N, and then 196−128 = 68, which is the ASCII code for D. This is where the word END is stored, then. The reason for treating the last letter so differently is to save memory! If a gap were left between words, this would be a byte of memory wasted. As it is, there is no waste, because the last letter of a group always has a code number that is greater than 128, so the computer can recognise it easily. We have followed the same scheme in the BASIC program of Fig. 1.4 by using line 30 to print the correct letter and to take a new line and print the address number.

Oric/Atmos cutaway

Now take a look at a diagram of the Oric/Atmos in Fig. 1.5. It's quite a simple diagram because I've omitted all of the detail, but it's enough to give us a clue about what's going on inside. This is the type of diagram that we call a 'block diagram', because each unit is drawn as a block with no details about what may be inside. Block diagrams are like small-scale maps which show us the main routes between towns but don't show side-roads or town streets. A block diagram is enough to show us the main paths for electrical signals in the computer.

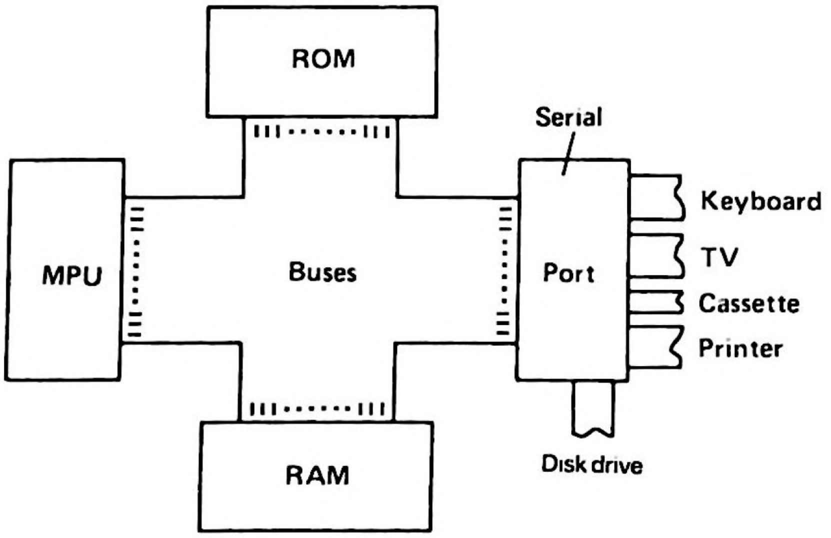


Fig. 1.5. A block diagram of Oric/Atmos. The connections marked 'Buses' consist of a large number of connecting links which join all of the units of the system.

The names of two of the blocks should be familiar already, ROM and RAM, but the other two are not. The block that is marked MPU is a particularly important one. MPU means Microprocessor Unit – some block diagrams use the letters CPU (Central Processing Unit). The MPU is the main 'doing' unit in the system, and it is, in fact, one single unit. The MPU is a single plug-in chunk, one of these silicon chips that you read about, encased in a slab of black plastic and provided with 40 connecting pins that are arranged in two rows of 20 (Fig. 1.6). There are several different types of MPU made by different manufacturers, and the one in your Oric/Atmos is called 6502 (or 6502A). The 6502A is a version of the 6502 which operates at a higher speed.

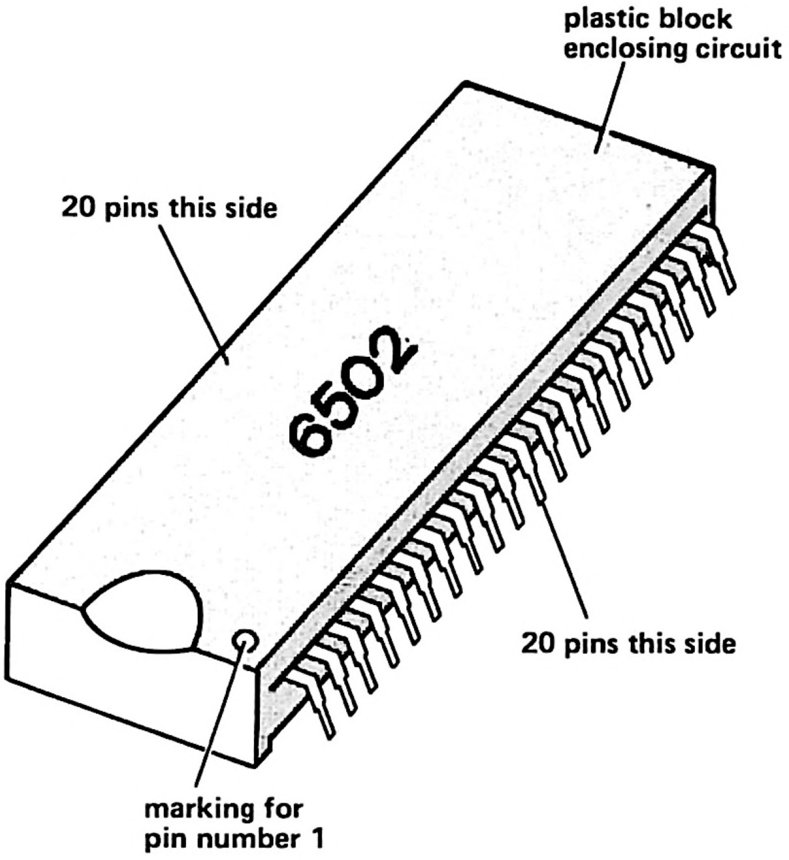


Fig. 1.6. The 6502 MPU. The actual working part is smaller than a fingernail, and the larger plastic case (52 mm by 14 mm wide) makes it easier to work with.

What does the MPU do? The answer is practically everything, and yet the actions that the MPU can carry out are remarkably few and simple. The MPU can load a byte, meaning that a byte which is stored in the memory can be copied into another store in the MPU. The MPU can also store a byte, meaning that a copy of a byte that is stored in the MPU can be placed in any address in the memory.

These two actions (see Fig. 1.7.) are the ones that the MPU spends most of its working life in carrying out. By combining them, we can copy a byte from any address in memory to any other. You don't think that's very useful? That copying action is just what goes on when you press the letter H on the keyboard and see the H appear on the screen. The MPU treats the keyboard as one piece of memory and the screen as another, and copies bytes from one to the other as you type. That's a considerable simplification, but it will do for now just to show how important the action is.

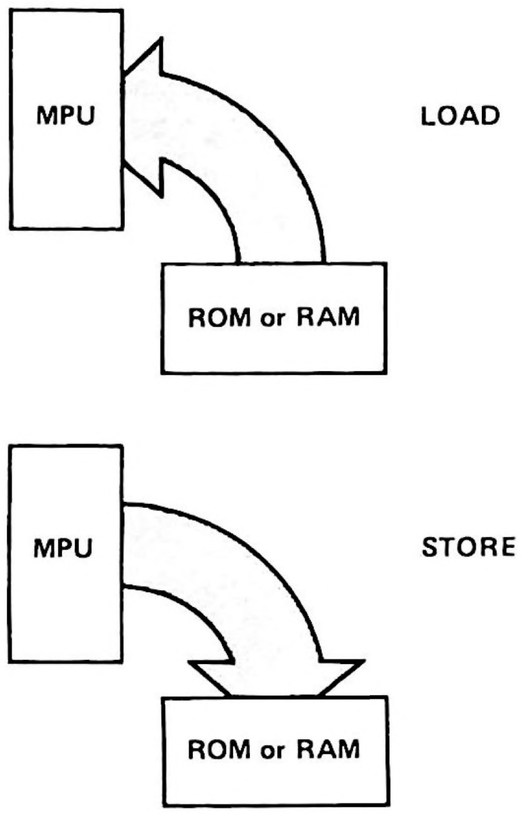


Fig. 1.7. Loading and storing. Loading means signalling to the MPU from the memory, so that the digits of a byte are copied into the MPU. Storing is the opposite process.

Loading and storing are two very important actions of the MPU, but there are several others. One set of actions is the 'arithmetic set'. For most types of MPU, these consist of addition and subtraction only, using only single-byte numbers. Since a single-byte number means a number between Ø and 255, how does the computer manage to carry out actions like multiplication of large numbers, division, raising to powers, logarithms, sines, and all the rest? The answer is by machine code programs that are stored in the ROM. If these programs were not there you would have to write your own. There aren't many computer users who would like to set about that task.

There's also the logic set. MPU logic is, like all MPU actions, simple and subject to rigorous rules. Logic actions compare the bits of two bytes and produce an 'answer' which depends on the bit values that are being compared and on the logic rule that is being followed. The three main logic rules are called AND, OR and XOR, and Fig. 1.8 shows how they are applied.

AND

The result of ANDing two bits will be 1 if both bits are 1, but ∅ otherwise:

$$1 \text{ AND } 1 = 1 \qquad \left\{\begin{matrix} 1 \text{ AND } \emptyset = \emptyset \\ \emptyset \text{ AND } 1 = \emptyset \end{matrix}\right\} \qquad \emptyset \text{ AND } \emptyset = \emptyset$$

For two bytes, corresponding bits are ANDed:

$$\begin{array}{rl} & 1\emptyset11\emptyset111 \\ \text{AND} & \underline{\emptyset\emptyset\emptyset\emptyset1111} \\ & \emptyset\emptyset\emptyset\emptyset\emptyset\underbrace{111} \end{array}$$

only these bits
exist in both bytes.

OR

The result of ORing two bits will be 1 if *either* or *both* bits are 1, but ∅ otherwise:

$$1 \text{ OR } 1 = 1 \qquad \left\{\begin{matrix} 1 \text{ OR } \emptyset = 1 \\ \emptyset \text{ OR } 1 = 1 \end{matrix}\right\} \qquad \emptyset \text{ OR } \emptyset = \emptyset$$

For two bytes, corresponding bits are ORed:

$$\begin{array}{rl} & 1\emptyset11\emptyset111 \\ \text{OR} & \underline{\emptyset\emptyset\emptyset\emptyset1111} \\ & 1\underset{\uparrow}{\emptyset}111111 \end{array}$$

only bit which
is ∅ in both.

XOR (Exclusive OR)

Like OR, but result is zero if the bits are identical:

$$1 \text{ XOR } 1 = \emptyset \qquad \left\{\begin{matrix} 1 \text{ XOR } \emptyset = 1 \\ \emptyset \text{ XOR } 1 = 1 \end{matrix}\right\} \qquad \emptyset \text{ XOR } \emptyset = \emptyset$$

$$\begin{array}{rl} & 1\emptyset11\emptyset111 \\ \text{XOR} & \underline{\emptyset\emptyset\emptyset\emptyset1111} \\ & \underbrace{1\emptyset111\emptyset\emptyset\emptyset} \end{array}$$

if two bits are identical
the result is zero.

Fig. 1.8. The rules for the three logic actions, AND, OR and XOR.

Another set of actions is called the 'jump set'. A jump means a change of address, rather like the action of GOTO in BASIC. A combination of a test and a jump is the way that the MPU carries out its decision steps. Just as you can program in BASIC:

```
IF A = 36 THEN GOTO 1Ø5Ø
```

so the MPU can be made to carry out an instruction which is at an entirely different address from the normal next address. The MPU is a programmed device, meaning that it carries out each of its actions as a result of being fed with an instruction byte which has been stored in the memory. Normally when the MPU is fed with an instruction from an address somewhere (usually in ROM), it carries out the instruction and then reads the instruction byte that is stored in the next address up. A jump instruction would prevent this from happening and would, instead, cause the MPU to read from another address – the one that was specified in the jump instruction. This jump action can be made to depend on the result of a test. The test will usually be carried out on the result of the previous action – whether it gave a zero, positive or negative result, for example.

That isn't a very long or exciting list, but the actions that I've omitted are either unimportant at this stage, or not particularly different from the ones in the list. What I want to emphasise is that the magical microprocessor isn't such a very smart device. What makes it so vital to the computer is that it can be programmed and that it can carry out its actions very quickly. Equally important is the fact that the microprocessor can be programmed by sending it electrical signals.

These signals are sent to eight pins, called the 'data pins', of the MPU. It doesn't take much of a guess to realise that these eight pins correspond to the eight bits of a byte. Each byte of the memory can therefore affect the MPU by sharing its electrical signals with the MPU. Since this is a long-winded description of the process, we call it 'reading'. Reading means that a byte of memory is connected along eight lines to the MPU, so that each 1 bit will cause a 1 signal on a data pin, and each Ø bit will cause a Ø signal on a data pin. Just as reading a paper or listening to a recording does not destroy what is written or recorded, reading a memory does not change the memory in any way, and nothing is taken out. The opposite process of writing does, however, change the memory. Like recording a tape, writing wipes out whatever existed there before. When the MPU writes a byte to an address in the memory, whatever was formerly stored at that address is no longer there – it has been replaced by the new byte. This is why it is so easy to write new BASIC lines replacing old ones at the same line number.

Table d'Hôte?

Do you really write programs in BASIC? It might sound like a silly question, but it's a serious one. The actual work of a program is done by coded instructions to the MPU, and if you write only in BASIC, you don't write these. All that you do is to select from a menu of choices that we call the BASIC keywords, and arrange them in the order that you hope will produce the correct results. Your choice is limited to the keywords that are designed into the ROM. We can't alter the ROM, and if we want to carry out an action that is not provided for by a keyword, we must either combine a number of keywords (a BASIC program) or operate directly on the MPU with number codes (machine code). When you have to carry out actions by combining a number of BASIC commands, the result is clumsy, especially if each command is a collection of other commands. Direct action is quick, but it can be difficult. The direct action that I am talking about is machine code, and a lot of this book will be devoted to understanding this 'language' which is difficult just because it's simple!

Take a situation which will illustrate this paradox. Suppose you want a wall built. You could ask a builder. Just tell him that you want a wall built across the back garden, and then sit back and wait. This is like using BASIC with a command word for 'build a wall'. There's a lot of work to be done, but you don't have to bother about the details.

Now think of another possibility. Suppose you had a robot which could carry out instructions mindlessly but incredibly quickly. You couldn't tell it to 'build a wall' because these instructions are beyond its understanding. You have to tell it in detail, such as: 'stretch a line from a point 85 feet from the kitchen edge of the house, measured along the fence southwards, to a point 87 feet from the lounge end of the house measured along that fence southwards. Dig a trench eighteen inches deep and one foot wide along the path of your line. Mix three bags of sand and two of cement with four barrow-loads of pebbles for three minutes. Mix water into this until a pail filled with the mixture will take ten seconds to empty when held upside down. Fill the trench with the mixture ...'. The instructions are very detailed – they have to be for a brainless robot – but they will be carried out flawlessly and quickly. If you've forgotten anything, no matter how obvious, it won't be done. Forget to specify how much mortar, what mixture and where to place it, and your bricks will be put up without mortar. Forget to specify the height of the wall, and the robot will keep piling one layer on top of another, like the Sorcerer's Apprentice, until someone sneezes and the whole wall falls down.

The parallel with programming is remarkably close. One keyword in BASIC is like the 'build a wall' instruction to the builder. It will cause a lot of work to be done, drawing on a lot of instructions that are not yours – but it may not be done as fast as you like. If you can be bothered with specifying the detail, machine code is a lot faster because you are giving your instructions direct to an incredibly fast but mindless machine, the microprocessor. We can stretch the similarity further. If you said to your builder 'mend the car', he might be unwilling or unable to do so. The correct set of detailed instructions to the robot would, however, get this job done. Machine code can be used to make your computer carry out actions that are simply not provided for in BASIC, though it's fair to say that many modern computers allow a much greater range of commands than early models, and this aspect of machine code is not quite so important as it used to be.

One last look at the block diagram is needed before we start on the inner workings of the Oric/Atmos. The block which is marked 'Port' includes more than one chip. A *port* in computing language means something that is used to pass information, one byte at a time, into or out from the rest of the system – the MPU, ROM and RAM. The reason for having a separate section to handle this is that inputs and outputs are important but slow actions. By using a port we can let the microprocessor choose when it wants to read an input or write an output. In addition, we can isolate inputs and outputs from the normal action of the MPU. This is why nothing appears on the screen in a BASIC program except where we have a PRINT command in the program. It's also why pressing the PLAY key of the cassette recorder has no effect until you type CLOAD (with a filename) and press ENTER. The port keeps the action of the computer hidden from you until you actually need to have an input or an output.

We have now looked at all of the important sections of your Oric/Atmos. I've used some terms loosely – purists will object to the way I've used the word 'port', for example – but no-one can quarrel with the actions that are carried out. What we have to do now is to look at how the computer is organised to make use of the MPU, ROM, RAM and ports so that it can be programmed in BASIC and can run a BASIC program. It looks like a good place to start another chapter!

Chapter Two
Digging Inside the Oric/Atmos

I don't mean digging inside literally – you don't have to open up the case. What I do mean is that we are going to look at how the Oric-1 and Atmos are designed to load and run BASIC programs. We'll start with a simplified version of the action of the whole system, omitting details for the moment.

The ROM of your Oric/Atmos, which starts at address 49152, consists of a large number of short programs – *subroutines* – which are written in machine code, along with sets of values (tables) like the table of keywords. The ROM of the Atmos is similar to that of the Oric-1, but there are several important differences. There will be at least one machine code subroutine for each keyword in BASIC, and some of the keywords may require the use of many subroutines. When you switch on your Oric/Atmos, the piece of machine code that is carried out first is called the 'initialisation routine'. This is a long piece of program, but because machine code is fast, carrying out instructions at the rate of many thousands per second, you see very little evidence of all this activity. All that you notice is the bars on the screen in the time between switching on and seeing the Oric copyright notice. In this brief time, though, the action of the RAM part of the memory has been checked, some of the RAM has been 'written' with bytes that will be used later, and most of the RAM has been cleared for use. 'Cleared for use' does not mean that *nothing* is stored in the RAM. When you switch off the computer, the RAM loses all trace of stored signals, but when you switch on again the memory cells don't remain storing zeros. In each byte, some of the bits will switch to 1 and some will switch to Ø when power is applied. This happens quite at random, so that if you could examine what was stored in each byte just after switching on, you would find a set of meaningless numbers. These would consist of numbers in the range Ø to 255, the normal range of numbers for a byte of memory. These numbers are 'garbage' – they weren't put into memory deliberately, nor do they form useful instructions or data. The first job of the computer, then, is to clean up. In place of the random numbers, the computer substitutes a very much more ordered pattern of bytes of number 85. Try this – switch on, and type (no line number):

```
FOR N=128Ø TO 148Ø:?PEEK(N);" ";:NEXT
```

and then press RETURN. The range of memory addresses we have used is the 'start of BASIC' range, where the first bytes of a BASIC program are normally stored. If we have just switched on, and haven't used a line number for the command, there will be nothing stored here except for the pattern that was left after the initialisation and as a result of the command. As you'll see on the screen, the pattern consists mainly of the chain of 85s. At the start, though, you will find other numbers. The numbers at the start are placed there by the FOR N = 128Ø ... command that you used to discover the pattern, and the other numbers are put there by the action of this command as it is executed. Why 85? Well, in binary, 85 is Ø1Ø1Ø1Ø1, and it's a good pattern to use to check if the memory is working correctly.

The initialising program has a lot more to do. The first section of RAM, from address Ø to 1279, is for 'system use'. This is because the machine code subroutines which carry out the actions of BASIC need to store quantities in memory as they are working. Address numbers 154 and 155, for example, hold the address of the first byte of a BASIC program. A much larger section of the memory is used for storing numbers that make text and graphics appear on the screen. In addition, some RAM has also to be used to hold quantities that are created when a program runs That's what we are going to look at now.

Variables on the table

BASIC programs make a lot of use of variables, meaning the use of letters to represent numbers and words. Each time you 'declare a variable' by using a line like:

```
N = 2Ø or A$ = "SMITH"
```

the computer has to take up memory space with the name (N or A$ or whatever you have used) and the value (like 2Ø or SMITH) that you have assigned to it. The piece of memory that is used to keep track of variables is called the variable list table (VLT). It doesn't occupy any fixed place in the memory, but is stored in the free space just above your program. If you add one more line to your program, the VLT address has to be moved to a set of higher address numbers. If you delete a line from your program, the VLT will be moved down in the same way so that it is always kept just following the last line of BASIC.

Now because the variable list table address can and does move around as the program is altered, the computer must at all times keep a note of where the table starts. This is done by using two bytes of a piece of memory that is reserved for system use, the addresses 156 and 157. You may wonder why two addresses are used. The reason is that one byte can hold a number only up to 255 in value. If we use two bytes however, we can hold the number of 256's in one byte and the remainder in the other. A number like 257, for example, is one 256 and one remaining. We could code this as 1,1. This means that a 1 is stored in the byte that is reserved for 256's, and 1 in the byte reserved for units. The order of storing the numbers is low-byte then high-byte. To find the number that is stored, we multiply the second byte by 256 and add the first byte. For example, if you found 3,8 stored in two consecutive addresses that were used in this way, this would mean the number:

```
8*256 + 3 = 2Ø51
```

The biggest number that we can store using two bytes like this is 255,255, which means 255*256 + 255 = 65535. This is the reason that you can't use very large numbers like 7ØØØØ as line numbers in the Oric/Atmos – the operating system uses only two bytes to store its line numbers. In fact, for other reasons, the maximum number that you can use is 63999.

All of this means that we can find the address that is stored in addresses 156 and 157 by using the formula:

```
?PEEK(156)+256*PEEK(157)
```

If you use this just after you have switched on your Oric/Atmos, then the result on the Oric/Atmos is the address number 1283. This is just above the address at which the first byte of a BASIC program would be stored. To see this in action, type the line:

```
1Ø N= 2Ø
```

and try ?PEEK(156)+256*PEEK(157) again. If you typed this line as I did, with a space between the line number and the 'N', then the address that you get is 1292. The variable list table has moved upwards in the memory, by 9 bytes. That's more than the number of bytes that you typed, you'll notice – reasons later.

Quite a lot of important addresses that the computer uses are 'dynamically allocated' like this. 'Dynamically allocated' means that the computer will change the place where groups of bytes are to be kept. It will then keep track of where they have been stored by altering an address that is held in a pair of bytes such as this example. This has important implications for how you use your computer. For example, if you shift the VLT by poking new numbers into addresses 156 and 157, the computer can't find its variable values. Try this – after finding the VLT address, but without running the one-line program, 1∅ N = 2∅ – type ?N. The answer will be zero. Why? Because the program has not been run. The address 1292 is where the VLT will start, but there's no VLT created until the program runs. This makes it easy for you to add or delete lines at this stage. All that will have to be altered is the pair of numbers in addresses 156 and 157. The VLT values are put in place only when the program runs, and a new table is created all over again each time you type RUN and press RETURN. Each time a new table is created, a new pair of bytes will be put into 156 and 157. That's why you can't resume a program after editing – you have to RUN again to create a new VLT at a new address. If you RUN the one-line program now, and then type ?N you will get the expected answer of 2∅. Now type (no line number) POKE 157,6, and press RETURN. This has changed the address of the VLT to an address where there is no VLT. Try ?N and see what you get. On my machine, it was zero, because the correct value of variable N can be found no longer. If your Oric/Atmos locks up during this exercise, then you may have to switch off to regain control. This seldom happens but if you do have to switch off, the program will be lost. Note, incidentally, the use of POKE to place a new value into a memory address. The correct form of the command is POKE A,D. A is an address, and will be in the range ∅ to 49151 (the range of values of RAM memory) for the 48K Oric/Atmos. D is the data that you place into this memory address, and it must be a value between ∅ and 255. If you try to POKE a number greater than 255, you will get an 'ILLEGAL QUANTITY ERROR' message instead.

A look at the table

It's time now to do something more constructive, and take a look at what is stored in the VLT. When we do these investigations, it's important to ensure that the computer is clear of the results of previous work, so it's advisable to switch off and then on again, before each effort. Simply pressing the RESTORE button under the computer does not alter values that you may have poked into the memory. It's tedious, I know, but that's machine code for you!

To work, then. After switching off and on again, type the line

```
1Ø N = 2Ø
```

again, and find the VLT address by using ?PEEK(156)+256*PEEK(157). This gave me the address 1292 again. Now type RUN, so that values are put into the VLT, and take a look at what has been stored there. This is done by using the command:

```
FOR X = 1292 TO 13ØØ:?X;" ";PEEK(X):NEXT
```

and pressing RETURN. This gives the listing that is illustrated in Fig. 2.1. Now can we recognise anything here? We ought to recognise the first byte of 78, because that's the ASCII code for N! The next byte is zero because our variable is called N, not N1 or NG or any other two-letter name. If we used a two-letter name, then both addresses 1292 and 1293 would have been occupied. The next five bytes, then, must be the way that the number 2Ø has been coded.

```
1292    78
1293    0
1294    133
1295    32
1296    0
1297    0
1298    0
1299    88
1300    0
```

Fig. 2.1. The variable list table entry for a simple number variable.

At this point, don't worry about how these numbers are used to represent 2Ø – just accept that they do! How do I know that it's the next five bytes that represent the number 2Ø? Easy, the byte in address 1299 is 88, which is the ASCII for X, and that's the variable that we used to print the table values! The Oric/Atmos always uses just five bytes for any value of number variable, no matter whether it's a small number like 2Ø, or a very much larger one like 1427Ø68315, or a fraction, or negative. This makes the storage of number variables simple, and it also makes it easy for the computer to find variables. If, for example, it is looking for the value of a variable called Y, then when it finds 'N' (coded as ASCII 78) it need not waste time with the next six bytes (one for a second letter, five for the value), but move to the next place where a variable name will be stored. If you are curious, and have a head for mathematics, Appendix A shows what method of coding is used to convert numbers into five bytes. For the purposes of this book you don't, however, need to understand how the coding is done as long as you know how the code is stored and how many bytes are needed.

Tying up a string

Now we need to take a look at how a string variable is stored. Switch off and on again, and then type the line:

```
1Ø ABS$ = "THIS IS A STRING"
```

RUN this one-liner, and then find the VLT address by using addresses 156 and 157 as before. I obtained 131Ø for this address. Now use:

```
FOR X = 131Ø TO 132Ø:?X;" ";PEEK(X):NEXT
```

to find what is in the VLT. This time, it's as Fig. 2.2 shows. The first value in this table is 65, which is the ASCII code for A. The second, however, is 194. Now this is the ASCII code for B with 128 added to it, and it's the way that the Oric/Atmos recognises that this is a string variable. If you had used the variable name A$ rather than AB$,

```
1310    65
1311    194
1312    16
1313    10
1314    5
1315    0
1316    0
1317    88
1318    0
1319    139
1320    37
```

Fig. 2.2. The VLT entry for a string variable.

then the second number (at address 1311) would have been 128, not Ø. When you use a number variable, the second ASCII code of the name will be Ø, or one of the ASCII code numbers, never greater than 127. Good thinking, designers!

Now take a look at the rest of the entry for this string. It doesn't look much like the ASCII codes for the letters, does it? In fact, the entry consists of seven bytes only, just the same total length as a number variable. The clue to what is being done emerges when we take a look at the numbers. The number that follows the code for B (194, because 128 has been added to the ASCII code) is 16. Now 16 is the number of characters in the string. If you count the number of letters and spaces you'll see that this is what it comes to. The next two bytes are 1Ø and 5. Now two bytes together are always likely to be an address, and if we combine them in the usual way, using 5*256+1Ø, we get 129Ø. The next step in the trail is to try PRINTPEEK(129Ø). Sure enough, it's 84, which is the ASCII code for 'T'. 129Ø, then, is the address of the first byte of the string.

Let's gather all this up. The Oric/Atmos stores an entry of seven bytes in its VLT for each string. Of these seven bytes, the first two are for the string name, and the second will be 128 or more. When a two-character name is used, 128 is added to the ASCII code for the second letter. This allows the computer to distinguish a string variable from a number variable. The next five bytes then contain the length of the string and the address in memory of its first byte. As it happens, only three bytes are needed to keep track of a string. One byte is needed for the length – no string will exceed 255 characters. Though you cannot enter a string of more than 8Ø characters from the keyboard, you can join strings to make a length of up to 255 characters. Two bytes are needed for the address, so that two of the seven bytes that are used in the string VLT entry are not needed except as separators. The convenience of having the same total length of VLT entry for a string as for a number outweighs the slight waste of the last two bytes in each string entry.

In this example, the string is stored at an address lower than the VLT, in the 'BASIC text' part of the memory. This is the part of the memory which contains the program, and since the ASCII codes for the string are placed here when you type the program, it's as good a resting place as any. Numbers must be transferred to the VLT because they are not stored as ASCII codes. The question now is, what happens when a string is created which does not exist in the program? Switch off, then on again, and type:

```
10 A$ = "AB":B$ = "CD":C$ = A$ + B$
```

Now RUN this, and you will find that your VLT is longer, as you might expect. You will have to look at memory addresses from 1312 to 1335 this time. You will find the entries for A$ and B$, just as you would expect, giving addresses inside the program memory region,

```
1312    65
1313    128
1314    2
1315    9
1316    5
1317    0
1318    0
1319    66
1320    128
1321    2
1322    17
1323    5
1324    0
1325    0
1326    67
1327    128
1328    4
1329    252
1330    151
1331    0
1332    0
1333    88
1334    0
1335    139
```

Fig. 2.3. The VLT entry for a string which is not stored in the program part of memory.

as shown in Fig. 2.3. The variable C$, however, gives the bytes 252,151 for its address. This corresponds to an address of 389∅8 (it's 151*256 + 252, remember) for this string. We can take a look at these addresses. If you type:

```
FORX=389∅8TO38911:?X;" ";PEEK(X);" ";CHR$
(PEEK(X)):NEXT
```

then all will be revealed. The ASCII codes for letters ABCD are now stored here, and the use of CHR$ in the program reveals them.

Into integers

We've looked at the storage of numbers and of strings, but we should not forget that the Oric/Atmos allows two types of numbers to be stored. One of these number types is called 'real', and a variable for a real number uses a letter, or pair of letters, or letter and digit like A, AB or A2 to represent it. A real number is coded in the way that we have already seen, using a total of seven bytes. Of these, two are for the name, and five for the value. A real number can be positive, negative, or fractional, and its range of size can be from about 10^{38} to 10^{-39}. The integer numbers are whole numbers, no fractions allowed, and can range from −32768 to +32767. It's time to take a look at these numbers in the VLT.

Start, as usual, by switching off and then on again to clear the memory. Now type the one-liner:

```
10 A%=15:B%=300
```

and RUN this. Find the position of the VLT as usual by peeking addresses 156 and 157. My Atmos came up with 1300. Now type:

```
FOR X=1300TO1318:PRINTX;" ";PEEK(X):NEXT
```

and press RETURN. This will give you the sequence which is shown in Fig. 2.4.

```
1300   193
1301   128
1302   0
1303   15
1304   0
1305   0
1306   0
1307   194
1308   128
1309   1
1310   44
1311   0
1312   0
1313   0
1314   88
1315   0
1316   139
1317   36
1318   192
```

Fig. 2.4. The VLT entry for two integer numbers.

It's not the same as the real number sequence that we saw in Fig. 2.1. To start with, something has been done to the first byte, the one that should represent the variable name of A%. The byte is 193, which is just 65 + 128. It's the code for A with 128 added. The second byte is 128. This suggests that when we create an integer variable name, the machine adds 128 to *both* of the letters of the name. Can you see the pattern in all this? For a real number, the letters of the name use ASCII codes. For a string, the second code for the name is 128, or an ASCII code with 128 added. For an integer, 128 has been added to the first code as well. This is how the machine can distinguish between different letter variables. The signs % and $ are *not* stored in the memory!

We need now to look at how the numbers are stored, though. The bytes that follow the two 'variable name' bytes are Ø,15,Ø,Ø,Ø, and this seems to indicate that the number 15 has been stored unaltered. This is quite unlike the transformation that we saw carried out for the storage of a real number. The number 3ØØ, however, is stored as 1,44. Now could this be a two-byte storage system? We can try it – 256*1 + 44 = 3ØØ. The number has been stored as two bytes, with the high byte first, unlike the order that is used for line numbers or memory address numbers. The VLT entry is still of seven bytes, with three of them unused this time.

How does this explain the use of integers? Well, for one thing, we can explain the range of integer numbers. If we use only two bytes for storage, we can't store a number greater than the one which uses 255 stored in each byte. This number would be 255*256 + 255, which is 65536. Now we know that the range of integers is from −32768 to +32767 – and these numbers add up to 65535! Instead of making integer numbers cover just Ø to 65535, then, the Oric/Atmos covers the more useful range of −32768 to +32767. The reasons for this range are illustrated in more detail in Appendix A.

Using integers has several other side-effects. One is that integers are stored with perfect precision. When you declare A% = 17412, then that's the number in the store, not 17411.9999999. A 'real' number is almost always an approximation, and when we use 'real' numbers, we have to be prepared for some 'rounding'. You've probably met calculators that told you that the answer to a problem was 3.9999999 rather than 4.0. This is caused by these approximations in storing real numbers, and some calculators would round the figure to 4 while others would not. If you work on the Oric/Atmos using integers, these problems do not bother you. The other advantage of using integers is speed. Because an integer number is stored in two bytes, the computer can deal with it much more quickly than with a 'real' number which requires five bytes, and which needs more elaborate decoding. If you want speed, then, use integers – but you probably knew that already!

Program time

It's time now to look at how a program is stored in the memory of your Oric/Atmos. As before, we shall rely on PEEKs at parts of the memory to find out what is happening. The first thing we need to know, however, is where the bytes that form the address of the start of a program are stored. As it happens, they are stored at 154 and 155.

We can therefore start looking at a program as it exists in the memory. Type the program as shown in Fig. 2.5, but don't run it.

```
10 A=10
20 PRINT A
30 C$="ATMOS"
```

Fig. 2.5. A simple BASIC program.

Now type:

```
?PEEK(154)+256*PEEK(155)
```

and you will find the address at which the first byte of this program starts. In this example, my Atmos gave the address 1281. Now when you use the usual loop to print values of the PEEK numbers from this address onwards, you get the list as shown in Fig. 2.6. At first sight it looks like a stream of meaningless numbers, but when you look more carefully, you can see some pattern in it. As usual, the ASCII codes act as useful signposts. Five places along, for example, you can see the number 65, which is the ASCII code for 'A'. Since we know that the line is 'A = 1Ø', we can look for the rest of this line.

```
 0     78     0     139    10     5     10     0     65
   212    49     48     0     18     5     20     0
 186    32    65     0     33     5     30     0     6
7    36    212    34    65    84    77    79
 83    34    0     0     0
```

Fig. 2.6. The bytes that represent the program in memory.

The 1∅ is recognisable as 49 (ASCII '1') and 48 (ASCII '∅'), so that the number 212 must represent the '=' sign. Note that 212 is *not* the ASCII code for '='. Instead, it is one of those 'tokens' that I mentioned in Chapter 1. It's a token because the computer is required to carry out an action, not just store an ASCII code here. The ∅ that follows the 49 (ASCII '∅') marks the end of this line.

Now we have to grapple with the first four bytes. The first two are, as you might suspect from looking at them, an address. The 1∅,5 makes up the address 5*256 + 1∅, which is 129∅. What is this address? Why, it's the address of the first byte of the next line! This is how the operating system of the Oric/Atmos can pick out lines, and put them into the correct sequence, no matter what order you use to enter them. The final mystery is easily solved. Looking at the third and fourth bytes of each of the lines shows the sequence 1∅, 2∅, 3∅ – the line numbers. There are two bytes reserved for the line numbers because we want to have line numbers higher than 255. For line numbers smaller than 256, the second of these bytes – the more significant byte – is not used. The last byte of each line is ∅.

Now take a look at the other lines, as they appear stored in the memory. We have met the PRINT token of 186 before, and all the rest should be familiar by now. The only novelty is the end of the program. The last line ends with a ∅ as usual, but following it, in the place where the address bytes for the next line would be, is another pair of zeros. This is the marker that the computer uses for END.

We can carry out some interesting changes on a program like this. Suppose, for example, that we poke the addresses that are used to carry the line numbers. If you type:

```
    POKE1292∅,1∅:POKE13∅∅,1∅
```

and press RETURN, you will have placed the number 1∅ in each line number address for the lines 2∅ and 3∅. Now LIST and look at the result! It's a program of line 1∅'s. Contrary to what you might expect, this will RUN perfectly normally. The action of running, you see, depends on the 'next line' addresses being correct, not on how the lines are numbered. A program that has been altered in this way, however, is certainly not normal. Try, for example, using the screen editing system to change the second line A to B, so that the line reads PRINT B. Now RUN and then LIST. You will find that the previous first line has disappeared, and the new first line is PRINT A, with PRINT B as the second line! You can, however, record a program which has been altered in this way, and replay it normally. This is the germ of a method by which you can make a program difficult to alter.

Running the program

Now that we have looked at the way in which a program is coded and stored in the memory of the Oric/Atmos, we can give a bit of thought as to how it runs. This action is carried out by the most complicated part of the operating system, and it has to be given a starting address. This address comes, as you might expect, from the locations 154 and 155 which we have used. Suppose we go through the actions, omitting detail, of the three-line program of Fig. 2.5. At the first address in BASIC, the RUN subroutine will read the first two bytes, and store them temporarily. These bytes will be used in place of the 'start of BASIC' address when the next line is carried out. The line number bytes are then read and stored. Why? So that if there is a syntax error in the line, the computer will be able to print out the message: 'SYNTAX ERROR IN 1Ø' rather than 'SYNTAX ERROR SOMEWHERE'! The next byte is an ASCII code, and the computer will take this as being a variable name. In the old days, the word LET had to be used to 'declare a variable'. There was a token for LET, but in most types of modern machines, this token is not used. The difference is that a token is now put in automatically for the '=' sign when a letter immediately follows a line number or a colon. If you do type LET, then the same token is used.

Following the ASCII code for 'A', the special token for the '=' sign then causes a subroutine to swing into action. This one creates an entry in the variable list table, at the first available address, and puts the ASCII code for A in that place. The next address in the VLT is left blank – there's no second letter for this variable name. The number 1Ø is then read and converted to the special binary form, as noted in Appendix A. This set of bytes is also placed in the VLT as the entry for A. The next byte of the program is then read – it's Ø, meaning end of line, so that the address for the next line, which was read as the first action, is now placed into the microprocessor. The type of action that we have considered in detail for line 1Ø then repeats with line 2Ø. This time, more has to be done when the action token is read. Since this is the token for PRINT, the subroutine for PRINT must be called up. It will locate the address of the next vacant place on the screen. This is done by keeping a note of the address in a couple of bytes of RAM – read these bytes, and you have the address. The value of A is then found in the VLT, and the bytes converted back to ASCII code form. The codes are then placed, one by one, in the screen memory. Doing this causes the characters to become visible on the screen, because of another subroutine. Once again, the zero at the end of the line causes the next line number to be used. At the end of the third line, however, the 'next line' number is zero, and the program ends. The computer goes back to its waiting state, ready for another command.

It's not quite so simple as that description makes it sound, but the essentials are there. The important thing to realise is that there is a lot of action to be done, and it has to be done one step at a time. What makes BASIC slow is that each token calls up a subroutine, which has to be found. For example, if you have a program that consists of a loop like:

```
10 FOR N = 1 TO 50
20 PRINT N
30 NEXT
```

then the action of reading the PRINT token of 186, and finding where the correct subroutine is stored, will be carried out 50 times. There is no simple way of ensuring that the subroutine is located once and then just used 50 times. The kind of BASIC that you have on your Oric/Atmos is 'interpreted' BASIC which means that each instruction is worked out as the computer comes to it. If that means finding the address of the PRINT subroutine 50 times, so be it. The alternative is a scheme called *compiling*, in which the whole program is converted to efficient machine code before it is run. Compiling is done using another program, called a *compiler*. The use of a compiler for a BASIC program very greatly speeds up the running of the program, but it makes the program less easy to edit, because it converts the program into a different form. You can't win them all!

Chapter Three
The Microprocessor

In this chapter, we'll start to get to grips with the 6502 microprocessor of the Oric/Atmos. The microprocessor, or MPU is, you remember, the 'doing' part of the computer as distinct from the storing part (memory) or the input/output part (ports), so that what the microprocessor does will control what the rest of the computer does.

The MPU itself consists of a set of memory stores for numbers, but with a lot of organisation added. By means of circuits that are very aptly called 'gates', the way in which bytes are transferred between different parts of the MPU's own memory can be controlled, and it is these actions that constitute the addition, subtraction, logic and other actions of the MPU. Each of the actions is programmed. Nothing will happen unless an instruction byte is present in the form of a 1 or a Ø signal at each of the 8 data terminals, and these bytes are used to control the gates inside the MPU. What makes the whole system so useful is that because the program instructions are in the form of electrical signals on eight lines, these signals can be changed very rapidly. The speed is decided by another electrical circuit called a 'clock-pulse generator', or 'clock' for short. The speed that has been chosen as standard for the clock of the Oric/Atmos is very fast, so that something like a million operations can be carried out per second.

Machine code

The program for the MPU, as we have seen, consists of number codes, each being a number between Ø and 255 (a single-byte number). Some of these numbers may be instruction bytes which cause the MPU to do something. Others may be data bytes, which are numbers to add, or store or shift, or tokens, or just ASCII codes for letters. The MPU can't tell which is which – it simply does as it is instructed. It's up to the programmer to sort out the numbers and put them into the correct order.

The correct order, as far as the MPU is concerned, is quite simple. The first byte that is fed to the MPU after switching on or after completing an instruction, is taken as being an instruction byte. Now several of the 6502 instructions consist of just one byte, and need no data. Others may be followed by one or two bytes of data, and some instructions need two bytes. When the MPU reads an instruction byte, then, it analyses the instruction to find if the instruction is one that has to be followed by one or more other bytes. If, for example, the instruction byte is one that has to be followed by two data bytes, then when the MPU analyses the first byte, it will treat the next two bytes that are fed to it as being the data bytes for that instruction. This action of the MPU is completely automatic, and is built into the MPU. The snag is that the machine code programmer must work to the same rules, and get the program right. 100% correct is *just about* good enough. If you feed a microprocessor with an instruction byte when it expects a data byte or with a data byte when it expects an instruction byte, then you'll have trouble. Trouble nearly always means an endless loop, which causes the screen to go blank and the keys to have no effect. Even the RESTORE button can sometimes fail to break the Oric/Atmos out of such a loop, and the only remedy is to switch off. You will generally lose whatever program you had in store, so it's vitally important to save any machine code program, or a BASIC program that causes machine code actions (by using POKE) on tape before you use it.

What I want to stress at this point is that machine code programming is tedious! It isn't necessarily difficult – you are drawing up a set of simple instructions for a simple machine – but it's often difficult for you to remember how much detail is needed. When you program in BASIC, the machine's error messages will keep you right, and help to detect mistakes. When you use machine code, you're on your own, and you have to sort out your own mistakes. In this respect, a type of program called an 'assembler' helps considerably. We'll look at that point again later. In the meantime, the best way to learn about machine code is to write it, use it, and make your own mistakes. We'll start looking at how this is done, and we'll begin with the ways of writing the numbers that constitute the bytes of a machine code program.

Binary, denary and hex

A machine code program consists of a set of number codes. Since each number code is a way of representing the 1's and ∅'s in a byte, it will consist of numbers between ∅ and 255 when we write it in our normal scale of ten (denary scale). The program is useless until it is fed into the memory of the Oric/Atmos, because the MPU is a fast device, and the only way of feeding it with bytes as fast as it can use them is by storing the bytes in the memory, and letting the MPU help itself to them in order. You can't possibly type numbers fast enough to satisfy the MPU, and even methods like tape or disk are just not fast enough.

Getting bytes into the memory, then, is an essential part of making a machine code program work, and we shall look at methods in more detail later on. At one time, simple and very short programs would be put into a memory by the most primitive possible method, using eight switches. Each switch could be set to give a 1 or ∅ electrical output, and a button could be pressed to cause the memory to store the number that the switches represented, and then select the next memory address. Programming like this is just too tedious, though, and working with binary numbers of 1's and ∅'s soon makes you cross-eyed. Now that we have computers, it makes sense to use the computer itself to put numbers into memory, and an equally obvious step is to use a more convenient number scale.

Just what is a convenient number scale is a matter that depends on how you enter the numbers and how much machine code programming you do. The Oric/Atmos contains subroutines which convert the binary numbers in its memory to the form of denary numbers to print on the screen, and will also carry out the reverse action. When you use PEEK, the address that you want can be written in denary, and the result of the PEEK will be a number in denary, between ∅ and 255. When you use POKE, similarly, you can type both the address number and the byte to be poked in denary.

Serious machine code programmers, however, find the use of denary anything but convenient. A denary number for a byte may be one figure (like 4) or two (like 17) or three (like 143). A much more convenient code is the one called *hex* (short for hexadecimal) code. All one-byte numbers can be represented by just two hex digits. In conjunction with this, serious machine code programmers write their programs in what is called *assembly language*. This uses command words which are shortened versions of the names of commands to the MPU. Programs that are called *assemblers* then convert these command words into the correct binary codes.

Practically all assemblers show these codes on the screen in hex form rather than in denary. In addition, when you type data numbers, you will have to make use of hex code. 'Hexadecimal' means scale of sixteen, and the reason that it is used so extensively is that it is naturally suited to representing binary bytes. Four bits, half of a byte, will represent numbers which lie in the range ∅ to 15 in our ordinary number scale. This is the range of one hex digit (Fig. 3.1).

Hex	Denary	Hex	Denary
∅	∅	C	12
1	1	D	13
2	2	E	14
3	3	F	15
4	4	then	
5	5	1∅	16
6	6	11	17
7	7	to	
8	8	2∅	32
9	9	21	33
A	1∅	22	34
B	11	etc.	

Fig. 3.1. Hex and denary digits.

Since we don't have symbols for digits higher than 9, we have to use the letters A,B,C,D,E, and F to supplement the digits ∅ to 9 in the hex scale. The advantage is that a byte can be represented by a two-digit number, and a complete address by a four-digit number.

The number codes that are used as instructions have been designed in hex code, so that we can see much better how commands are related. For example, we may find that a set of related commands all start with the same digit when they are written in hex. In denary, this relationship would not appear. In addition, it's much easier to write down the binary number which the computer actually uses when you see the hex version. The use of the Oric/Atmos assembler and monitor programs, such as the Oricmon Editor/disassembler/assembler, demands familiarity with hex, and books of information on the 6502 MPU will all be written assuming that you know hex. It sounds as if we ought to make a start on it!

The hex scale

The hexadecimal scale consists of sixteen digits starting, as usual, with Ø and going up in the usual way to 9. The next figure is not 1Ø, however, because this would mean one sixteen and no units, and since we aren't provided with symbols for digits beyond 9, we use the letters A to F. The number that we write as 1Ø (ten) in denary is written as ØA in hex, eleven as ØB, twelve as ØC and so on up to fifteen, which is ØF. The zero doesn't have to be written, but programmers get into the habit of writing a data byte with two digits and an address with four even if fewer digits are needed. The number that follows ØF is 1Ø, sixteen in denary, and the scale then repeats to 1F (thirty-one) which is followed by 2Ø (thirty-two). The maximum size of byte, 255 in denary, is FF in hex.

When we write hex numbers, it's usual to mark them in some way so that we don't confuse them with denary numbers. There's not much chance of confusing a number like 3E with a denary number, but a number like 26 might be hex or denary. The convention that is followed by 6502 programmers is to use the dollar sign ($) to mark a hex number, with the sign placed before the number. For example, the number $47 means hex 47, but plain 47 would mean denary forty-seven. When you write hex numbers for a 6502 program, it's advisable to follow this convention.

Hex	Binary	Hex	Binary
Ø	ØØØØ	8	1ØØØ
1	ØØØ1	9	1ØØ1
2	ØØ1Ø	A	1Ø1Ø
3	ØØ11	B	1Ø11
4	Ø1ØØ	C	11ØØ
5	Ø1Ø1	D	11Ø1
6	Ø11Ø	E	111Ø
7	Ø111	F	1111

Fig. 3.2. Hex and binary digits.

Now the great value of hex code is how closely it corresponds to binary code. If you look at the hex-binary table of Fig. 3.2, you can see that $9 is 1ØØ1 in binary and $F is 1111. The hex number $9F is therefore just 1ØØ11111 binary – you simply write down the binary digits that correspond to the hex digits. The conversion in the opposite direction is just as easy – group the binary digits in fours, starting at the least significant (right-hand side of the number) and then convert each group into its corresponding hex digit. Figure 3.3 shows examples of the conversion in each direction so that you can see how easy it is.

Conversion: Hex to Binary
Example: 2CH
2H is ØØ1Ø binary
CH is 11ØØ binary
So 2CH is ØØ1Ø11ØØ binary (data byte)

Example: 4A7FH
4H is Ø1ØØ binary
AH is 1Ø1Ø binary
7H is Ø111 binary
FH is 1111 binary
So 4A7FH is Ø1ØØ1Ø1ØØ1111111 binary (an address)

Conversion: Binary to Hex
Example: Ø11Ø1Ø11
1Ø11 is BH
Ø11Ø is 6H
So Ø11Ø1Ø11 is 6BH

Example: 1Ø11Ø1ØØ1ØØ1Ø (note that this is not a complete number of bytes).
Group into fours, starting with lsb:
ØØ1Ø is 2H
1ØØ1 is 9H
Ø11Ø is 6H and the remaining 1 is 1, making 1692H

Fig. 3.3. Converting between hex and binary.

The Oric/Atmos has built-in programs for converting between denary and hex. These are forced into action by the use of two 'commands', the symbol # (hashmark) and the word HEX$. Taking these in order, # has the effect of converting a hex number into denary. If you want to poke a number into the memory of the Oric/Atmos you *must* use denary, because all the systems of the machine are geared to convert denary numbers into binary. The machine will therefore work happily with the number 162, taking this as a denary number, but it won't recognise that A2 is the same number. If you type #A2, however, the computer will convert this into denary 162 and work with it. This can be confusing, because the sign # is used in other ways, and you will have to be careful, later on, about how you use the sign. You must *not* use the $ sign in your hex numbers, even though it is often found in listings, because the machine cannot deal with it.

The opposite conversion, from denary to hex, is done by using a string, HEX$. HEX$ has to be followed by a number, within brackets, and the result will be the hex equivalent of that number. If, for example, you type:

```
    PRINT HEX$(162)
```

and press RETURN, you will see the result A2 on the screen, the hex equivalent of denary 162. Having these two conversion methods built into the computer makes it very easy to design programs that print hex numbers, or which allow you to input hex numbers.

There's just one little twist to the use of hex. Suppose you have a string variable, D$, which has been allocated to a hex number. How do you print its denary value? The answer is to use VAL and # to convert the number, first into a denary string, then into a number. For example, if D$ = "A2", then PRINT VAL("#" + D$) will give 162, the denary value. The importance of this is that if you keep a set of hex numbers in a DATA line, you don't have to use:

```
    DATA #A2,#11,#F3,#4A
```

and so on, but simply:

```
    DATA A2,11,F3,4A
```

which saves you quite a lot of typing. Just in case you want to do these conversions when you are not near a friendly Oric/Atmos, though, the methods are shown in Appendix B.

Assuming, which is reasonable, that you don't want to commit yourself to the cost of a full-scale assembler at this point, what do you do to create machine code programs? The answer is that you design your program in assembly language, which is by far the easiest way to design machine code programs, and then you convert into hex code. Converting means looking up – in a set of tables called the *instruction set* – the hex number that represents each instruction. Instruction sets are provided by the manufacturers of all microprocessors, and Mostek, who designed the 6502, provide one for this chip. Just to assist you, a quick-reference guide has been included in this book, in Appendix B. Don't refer to it at the moment – it'll put you off!

Negative numbers

Useful as these denary-hex conversion programs for the Oric/ Atmos are, they are deficient in one respect – they don't handle negative numbers. Now this is unfortunate, though understandable. Negative numbers are very important in machine code programs, particularly if you are working without an assembler. The reason is that you sometimes want the MPU to do the equivalent of a GOTO, perhaps jumping to a step which is 30 steps ahead of its present address. This sort of thing is usually programmed by supplying a data number which is the number of steps that you want to skip. If you want to jump back to a previous step, however, you will need to use a negative number for this data byte. This is very common, because it's the way that a loop is programmed in machine code. We need, therefore, to know how to write a negative number in hex.

What makes it awkward is that there is no negative sign in hex arithmetic. There isn't one in binary either. The conversion of a number to its negative form is done by a method called *complementing*, and Fig. 3.4 shows how this is done. At first sight, and very often at second, third, and fourth, it looks entirely crazy. When you are dealing with a single byte number, for example, the denary form of the number −1 is 255! You are using a large positive number to represent a small negative one! It begins to make more sense when you look at the numbers written in binary. The numbers that can be regarded as negative all start with a 1 and the positive numbers all start with a Ø. The MPU can find out which is which just by testing the left-hand bit, the most significant bit.

It's a simple method, which the machine can use efficiently, but it does have disadvantages for humans. One of these disadvantages is that the digits of a negative number are not the same as those of a positive number. For example, in denary, −4Ø uses the same digits as +4Ø. In hex, −4Ø becomes $D8 and +4Ø becomes $28. The denary number −85 becomes $AB and +85 becomes $55. The second disadvantage is that humans cannot distinguish between a single byte number which is negative and one which is greater than 127. For example, does $9F mean 159 or does it mean −97? The short answer is that the human operator doesn't have to worry. The microprocessor will use the number correctly no matter how we happen to think of it. The snag is that we have to know what this correct use is in each case. Throughout this book, and in others that deal with machine code programming, you will see the words 'signed' and 'unsigned' used. A signed number is one that may be

Binary. Number (8 bits):	∅∅11∅1∅1	(denary 53)
Step 1: change each ∅ to 1, each 1 to ∅	11∅∅1∅1∅	
Step 2: add 1	+1	
The result is the negative form of number	11∅∅1∅11	(denary 2∅3)
Denary. Number	53	
Step 1: subtract from 256	2∅3	
This is negative form (in denary)		
Hex. Number	$35	Remember that F represents 15 denary, and 15 − 5 = 1∅ denary, which is A in hex.
Step 1: subtract from $FF	FF	
	35	
	$CA	
Step 2: add 1	+1	This is easier than a subtraction from $1∅∅, which involves the use of carries in hex.
	$CB	
The result is the hex form of negative number		

An alternative is to do the denary conversion, then convert back to hex.

Fig. 3.4. The two's complement, or negative form, of a binary number.

negative or positive. For a single byte number, values of ∅ to $7F are positive, and values of $8∅ to $FF are negative. This corresponds to denary numbers ∅ to 127 for positive values and 128 to 255 for negative. Unsigned numbers are always taken as positive. If you find the number $9C described as signed, then, you know it's treated as a negative number (it's more than $8∅). If it's described as unsigned, then it's positive, and its value is obtained simply by converting. How do we convert a signed single-byte hex number into denary, using our programs? It's simple: if the number is greater than $7F, then subtract 256 from its denary value. If you get 24∅ as a result, for example, then 24∅−256=− 16, and that's the signed value in denary

Light relief

Just to take a break from all this arithmetic let's look at screen displays on the Oric/Atmos. Each part of the screen can be controlled by whatever is stored in part of the memory, but there are two varieties of this memory. The simplest one to work with is the part which is called 'text screen memory'. It takes up memory addresses $BBA8 to $BFDF, which is 48Ø4Ø to 49119 in denary. What we mean by 'text screen memory' is that this piece of memory is treated in a special way. Anything that is stored here will be used to display text on the screen. That means that any code number that you store at an address in this range will produce the corresponding letter or graphics shape on the screen. For example, if you clear the screen, using CTRL L, and then type

```
POKE48619,65
```

and then press RETURN, you will see the letter 'A' appear about the middle of the screen. In general, if you poke an ASCII code number to any of the correct screen addresses, you will see a letter appear.

It's not quite so simple as it all sounds, however, when you come to the first address for each column of the screen. Suppose we take 486ØØ, the first address in the line that we have already used. Try the effect of:

```
POKE486ØØ,19
```

It makes the line turn to a yellow background. The Oric/Atmos uses this first column in each line of the text screen to place codes that affect the background colour of the rest of the line. If you use the list of colour codes in Fig. 3.5, then you can find the number that you need to poke into this first column address by adding 16 to the colour number. The colour that you will see as the background colour of the line will be given by this colour number. Now try this one. Clear the screen again, and POKE48619,65 to get the letter 'A'. Now use POKE486Ø1,6. This will make the letter 'A' appear in the colour cyan. These foreground colour pokes use the colour numbers of Ø to 7 directly.

You can program screen poke operations into a loop, as Fig. 3.6 shows. You should restore normal screen conditions by using CTRL L first. The effect of this loop is to fill the screen with A's. The filling is not particularly fast, because we're using BASIC in the loop. Later, we'll look at the same sort of thing in machine code, which is stunningly fast! For the moment, though, look at the effect on this

Number	Colour
∅	Black
1	Red
2	Green
3	Yellow
4	Blue
5	Magenta
6	Cyan
7	White

Fig. 3.5. Foreground colour codes for the Oric/Atmos. You need to add 16 to the code number for background colour.

```
10 N1=48042:N2=48079:REPEAT
20 FORJ=N1TON2:POKEJ,65:NEXT
30 N1=N1+40:N2=N2+40
40 UNTIL N1=49122
```

Fig. 3.6. A program which fills the screen with the letter A.

program of typing LORES1, pressing RETURN, and then running the program. It's quite useful, and if you want to avoid the 'hole' in the pattern caused by the READY message, then add an endless loop to the program, like:

```
5∅ GOTO 5∅
```

Note, however, when you use this with LORES1, you will see the graphics characters on the main part of the screen, but the letter A in the 'text window' at the bottom of the screen.

Now for the cream topping. Change your POKE program so that N1 = 48∅4∅. This means that the code number of 65 will be poked into the first two blocks in each line. What effect does it have? Just a complete screenful of 'A', that's all! What's more, you still get letter 'A' whether you type LORES1 or LORES∅. Not quite so simple after all, is it? What it amounts to is that you get a character whenever you poke a character code, anywhere in the screen memory. Poking a colour foreground or background code will also affect the screen. If, for example, you POKE X,19 in line 2∅ of Fig. 3.6, you will see the whole screen turn yellow when you RUN. You would, however, get the same effect more economically by just poking the first column of each line. POKE, unlike PRINT, can affect any column of the screen, but it's only the first two columns which will affect all the rest of the screen. Now try making N1 = 48∅6∅. As you might expect, this affects the right-hand side of the screen, turning it yellow. Now clear the screen, and try the one-liner:

```
FORN = 48∅6∅TO491∅∅STEP4∅:POKEN,19:NEXT
```

Now watch! This makes the right half of the screen turn yellow, and very much more quickly than the previous effort. The reason, which you probably know from BASIC, is that setting a colour 'attribute' number into any memory location affects *all the rest of the line*. This is one of the features which make the Oric/Atmos such a delight to program. It also means that we have to be quite careful about what we poke, and where, on the screen!

Chapter Four
6502 Details

Registers – PC and accumulator

A microprocessor consists of sets of memories, which are called *registers*. These memories are of a rather different type compared to ROM or RAM. The registers are connected to each other and to the pins on the body of the MPU by the circuits that are called *gates*. In this chapter, we shall look at some of the most important registers of the 6502 and how they are used. A good starting point is the register which is called the PC – short for *Program Counter*.

No, it doesn't count programs – what it does is to count the steps in a program. The PC is a sixteen-bit (two-byte) register which can store a full-sized address number, up to $FFFF (65535 denary). Its purpose is to count the address number, and the number that is stored in the PC will be incremented (increased by 1) each time an instruction is completed, or when another byte is needed. For example, if the PC holds the address $1F3A (denary 7994), and this address contains an instruction byte, then the PC will increment to $1F3B (denary 7995) whenever the MPU is ready for another byte. The next byte will then be read from this new address.

What makes the PC so important is that it's the automatic way by which the memory is used. When the PC contains an address number, the electrical signals that correspond to the Ø's and 1's of that address appear on a set of connections, collectively called the *address bus*, which link the MPU to all of the memory, RAM and ROM. The number that is stored in the PC will select one byte from the memory, the byte which is stored at that address number. At the start of a read operation, the MPU will send out a signal called the read signal on another line, and this will cause the memory to connect up the selected parts to another set of lines, the *data bus*. The signals on the data bus then correspond to the pattern of Ø's and 1's that is stored in the byte of memory that has been selected by the address in the PC. Each time the number in the PC changes, another byte of memory is selected, so that this is the way by which the MPU can keep itself fed with bytes. When the MPU is ready for another byte, the PC increments, and another read signal is sent out.

There are other ways in which the PC number can be changed, but for the moment we'll pass over that and look at another register, the *accumulator*. The accumulator of a microprocessor is the main 'doing' register of the MPU. This means that you would normally use it to store any number that you wanted to transfer somewhere else, or add to or carry out any other operation upon. The name of accumulator comes from the way in which this register operates. If you have a number stored in the accumulator, and you add another number to it, then the result is also stored in the accumulator. The nearest equivalent in BASIC is using a variable A, and writing the line:

```
A = A + N
```

where N is a number variable. The result of this BASIC line is to add N to the old value of A, and make A equal this new value. The old value of A is then lost. The accumulator acts in the same way, with the difference that an accumulator can't store a number greater than 255 (denary).

The 6502 has one accumulator register, often labelled as the *A register*. The importance of this is that it is used much more than the other registers, because so many actions can be carried out more quickly, more conveniently, or perhaps *only*, in the accumulator. When we read a byte from the memory, we usually place it in the accumulator. When we carry out any arithmetic or logic action, it will normally be done in an accumulator and the result also stored in the accumulator.

Addressing methods

When we program in BASIC, we don't have to worry about memory addresses at all unless we are using PEEK or POKE. The task of finding where bytes are stored is dealt with by the operating system of the machine. When a variable is allocated a value in a BASIC program as, for example, by a line like:

```
1Ø N = 12
```

we never have to worry about where the number 12 is stored, or in what form it happens to be stored. Similarly, when we add the line:

```
20 K = N
```

we don't have to worry about where the value of N was stored or where we will store the value of K. Remembering our comparison with wall-building in Chapter 2, we can expect that when we carry out machine code programming we shall have to specify each number that we use or, alternatively, the address at which the number is stored. This way in which we obtain a number, or find a place to store it, is called the 'addressing method'. What makes the choice of addressing method particularly important is that a different code number is needed for each different addressing method for each command. This means that each command exists in several different versions, with a different code for each addressing method. A list of all the 6502 addressing methods at this stage would be rather baffling, and for that reason has been consigned to Appendix C. What we shall do here is to look at some examples of selected addressing methods and the way that we write them in standard assembly language.

Assembly language

Trying to write down machine code directly as a set of numbers is a very difficult process which is liable to errors from beginning to end. The most useful way of starting to write a program is to write it in a set of steps in what is called *assembly language* (or *assembler language*). This is a set of abbreviated command words, called *mnemonics*, and numbers which are the data or address numbers. The numbers can be in hex or in denary, provided they are supplied to the computer in the correct form. Each line of an assembly language program indicates one microprocessor action, and this set of instructions is later 'assembled' into machine code, hence the name.

The aim of each line of an assembly language program is to show the action and the data or address that is needed to carry out that action, just as when we make use of TAB in BASIC we need to complete the command with a number. The part of the assembly language that specifies what is to be done is called the *operator*, and the part which specifies what the action is done to or on is called the *operand*. A few instructions need no operand, and we'll look at some later.

An example makes this easier. Suppose we look at the assembly language line:

```
LDA #$12
```

The operator is LDA, a shortened version of LOAD A, meaning that the accumulator register A is to be loaded with a byte. The operand is #$12, of which the $12 means that this is 12 hexadecimal (eighteen), rather than twelve denary. The other mark, the hashmark #, is used to show the addressing method that is to be used, a method called 'immediate addressing'. This is where some confusion can arise, because the Oric/Atmos uses the hashmark to signal hex to denary conversion. Because of this, some assemblers for the Oric/Atmos use IM to mean immediate, rather than #. For the moment, though, we'll stick to # in writing assembly language, because that's what you'll see used in a lot of books that deal with 6502 programming. It's also the standard symbol that the manufacturers of the chip use.

The whole line, then, should have the effect of placing the number $12 into the accumulator register A. It is the equivalent in machine code terms of the BASIC instruction:

```
A = 18  (remember that $12 is denary 18)
```

You could imagine that the memory which held the number was inside the microprocessor rather than part of the RAM memory, and was labelled with the name of 'A'.

A command such as LDA #$12 is said to use *immediate* addressing, because the byte which is loaded into the accumulator must be placed in the memory byte whose address *immediately* follows that of the instruction byte. It's like leaving a note for your milkman that says 'money in envelope next door'! There is one code number for the LDA # part of the whole instruction, and this byte is $A9, so that the hex sequence in memory of A9 12 will represent the entire command LDA #$12. It's a lot easier to remember what LDA #$12 means that to interpret A9 12, however, which is why we use assembly language as much as possible.

Immediate addressing like this can be convenient, but it ties you down to the use of one definite number and one fixed memory address. It's rather like programming in BASIC:

```
N = 4*12 + 3
```

rather than

```
N = A*B+C
```

In the first example, N can never be anything else but 51, and we might just as well have written: N = 51. The second example is very much more flexible, and the value of N depends on what values we choose for the variables A, B and C. When a machine code program is held in RAM, then the numbers which are loaded by this immediate addressing method can be changed if we must change them, but when the program is held in ROM no change is possible – and that's just one reason for needing other addressing methods. One of these other methods is *absolute addressing*.

Absolute addressing uses a complete two-byte address as its operand. This creates a lot of work for the 6502, because when it has read the code for the operator, it will then have to read two more bytes to find the memory address at which the data is stored. It will then have to place this address in the PC, read in the data byte, carry out the operation, and then restore the next correct address into the PC. Figure 4.1 shows in diagram form what has to be done. An absolute-addressed operation is therefore a lot slower to carry out than an immediate one, but since any byte may be stored at the address which is specified, it's easy to alter the data.

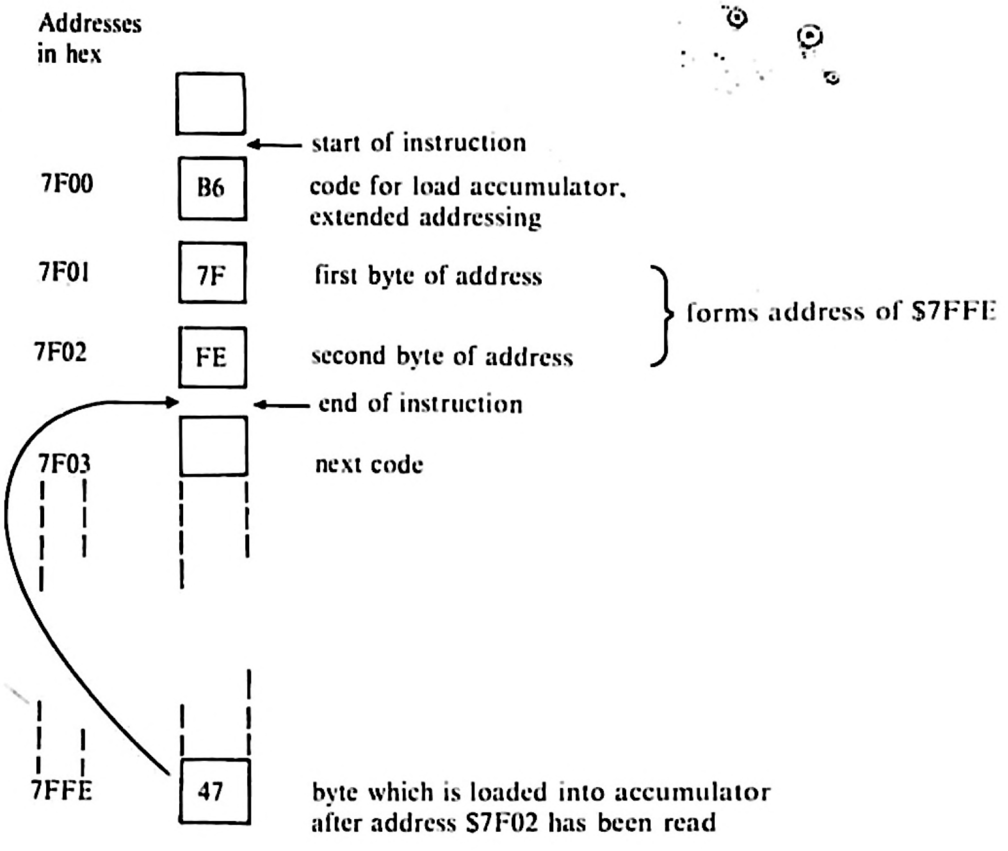


Fig. 4.1. How the absolute addressing method works.

Suppose, for example, that we have the instruction:

```
LDA $7FFE
```

In this slice of assembly language, the operator is LDA (load the accumulator) and the operand is the address $9620. What you have to remember is that what is put into the register A is not 9620, which is a two-byte address, but the data byte which is stored in memory *at this address*. The effect of the complete instruction, then, is to place a copy of the byte which is stored at $9620 into the accumulator A of the 6502. When the instruction has been completed, the address $9620 will still hold its own copy of the byte, because reading a memory does not change the content of the memory in any way.

We can also use the absolute addressing method in a command which will store a byte into the memory. The command:

```
STA $962Ø
```

means that the byte that is stored in the accumulator A is to be copied to memory at address $962Ø. This action will change the content of this memory address, but the accumulator A will still hold the same byte after the instruction has been carried out.

Zero page addressing

Zero page addressing is a method that allows you to specify a full address by using only one byte! The secret is that the upper byte of the address is taken to be zero. This is how we get the name of zero page for this type of addressing. Suppose, for example, that we used the command:

```
LDA $3F
```

There's no # sign here, so it does not mean that we load the number $3F into the accumulator. What it does mean is that we load the accumulator with the byte that has been stored in the address $ØØ3F. The ØØ part is the zero page, and the 3F part is the part that we have specified in the instruction. The range of addresses that we can use with this method is from $ØØØØ to $ØØFF only. This is only 256 (denary) bytes, but it's a very important 256 bytes. In denary, it's addresses Ø to 255 – you can see an advantage of using hex here, because if you looked only at the denary numbers, you would wonder why the words 'zero page' were used. In the 6502, zero page addressing allows us to get access to any of these addresses very rapidly, and with only one byte of address (the lower byte). This makes these addresses the favoured ones for any programmer who wants to carry out rapid loading and storing. For this reason, the 'zero page' addresses of the RAM in any 6502 machine are favourites for storing important quantities. We have already seen them being used for storing quantities like the address of the start of BASIC and the address of the Variable List Table. The same addresses are used for all sorts of important quantities, and as we go on in this book, we shall look at several of them in more detail. Because the Oric/Atmos makes a lot of use of zero page addresses for its own purposes, we have to be careful how we make use of them in our own programs. If we try to use an address that the computer needs, we may lock up the whole system. That's just one reason why it's so important to record a machine code program before you try it out.

Indexed addressing

Indexed addressing is a method which is particularly useful on the 6502. The principle is that an eight-bit register is used to hold a byte. This byte is then added to a base address when the indexed addressing is used. A base address means an address to which we can add a number before using it. For example, suppose that we had the number $4C stored in an index register. If we then specify that we want to load from an address $7ØØØ with indexing, then what happens is that the number in the index register is added to the address $7ØØØ. This makes the address number $7Ø4C, and this is the number that will be used as the address number. The effect is that the byte in address $7Ø4C will be loaded into the accumulator.

There are two registers in the 6502 that can be used in this way, the X and Y registers. They are both eight-bit registers, and they are almost identical, but there are some differences that will be more important to you later on. One of the differences that we can look at immediately is *zero page indexed addressing*. When we load the X register with a byte such as $4A, and then issue the command (in assembly language):

```
LDA $7ØØØ,X
```

this means that the address that is to be used is $7ØØØ plus the contents of the X register. The result is that the address $7Ø4A is used. This is *absolute indexed addressing*. 'Absolute' means that we are using absolute addressing for the 'base address' of $7ØØØ, and we are then adding the 'index' number of 4A from the X index register. We can use the Y index register of the 6502 in exactly the same way, with instructions such as:

```
    LDA $7ØØØ,Y
```

in assembly language. The use of the X index register, however, allows us to use zero page indexed addressing. We can, for example, use assembly language commands such as:

```
    LDA $ØC,X
```

This means that the base address is $ØØØC, and that the number stored in the X register will be added to $ØØØC before use. If the number stored in the X register is $23, then $ØØØC + $23 gives $ØØ2F, and this is the address number that will be used. The accumulator will therefore be loaded from address $ØØ2F. Zero page indexed addressing cannot be used with the Y-index register, so a command such as:

```
    LDA $1F,Y
```

is impossible – there is no instruction code for it.

Now the use of indexed addressing might not seem particularly useful at first sight. All you are doing, after all, is to add a number to an address. What makes the method so very useful is that you can alter the number that is stored in the X or Y registers. More particularly, you can increment or decrement the number in either of these index registers. The command INX means 'increment X'. Its effect is to add one to the number that is stored in the X register. Now since the number in the X register is added to a 'base address' when we use indexed addressing, incrementing X means that we shall increment the address that we get from an X-indexed load or store operation. Suppose, for example, that you wanted to store ten bytes at ten consecutive addresses. It's a very common problem, because it's just what you need to do to print ten characters on the screen, for example. The use of indexing means that you can set up a loop which will store, using indexing, then increment the index and repeat the store operation. Carry this out for a total of ten times, and the action is complete. Perhaps it's a bit early to mention the use of a loop, but we'll soon come to it. The example is a useful one, because it illustrates one of the most common uses for indexed addressing.

In addition to incrementing the X and Y registers (mnemonics INX, INY), we can decrement these registers. The assembly language command DEX means decrement X (subtract 1 from the number stored in X), and DEY means decrement Y. Since we have two index registers it's even possible to load from one base address, X-indexed, and then increment X. The byte that has been loaded can then be stored, indexed to the Y register this time and using another base address. The Y register can then be incremented. If all this is done in a loop, the effect will be to place bytes from a set of addresses, moving up from a starting address, into another set of addresses, moving up from another starting address. Complicated, you think? It might sound so in words, but in fact it's a very simple and neat method of shifting bytes from one part of memory to another, and that's an operation which is very often essential in computing. It can, for example, allow you to shift a BASIC program from one part of memory to another so that you can load in another program without losing the BASIC one.

Indirect addressing

Indirect addressing means going to an address to pick up another address at which a byte is located. It's like going to the address of a tourist office to find the address of a hotel (for a quick byte?). The 6502 allows two main forms of indirect addressing to be used. These are relatively complicated, and we won't make much use of them in this book, because this is an introduction, not an encyclopaedia. We can, however, look at the principles that are involved, because the two indirect methods are similar in many respects. The principle is to make use of the zero page addresses in pairs. In any of these pairs, we can store an address in two bytes. The order of the bytes is the one that should be familiar to you by now, low-byte and then high-byte. An indirect addressing method makes use of the first of a pair of these zero page addresses. The effect of a command which makes use of indirect addressing is therefore to read the byte in the first of the zero page addresses, and place this into the lower half of the Program Counter register. The next zero page address is then read, and the byte in it is put into the higher half of the Program Counter. The Program Counter now contains a full address, and this is used for loading or storing, depending on which operation has been specified. For example, if address $1Ø contained $ØØ and address $11 contained $3Ø, then the effect of an indirect load from $1Ø would be to place the address $3ØØØ into the Program Counter, and so load the accumulator with the byte that is stored at $3ØØØ. Once again, it seems very complicated and unnecessary until you realise the special advantages of such a method. The special advantage is that you can alter the address that is used by altering numbers stored in the zero page memory. The operating system of the Oric/Atmos makes use of indirect addressing along with the addresses that are stored in its page zero part of RAM. The indirect addressing methods of the 6502 are, in fact, rather more complicated than I have indicated here, because the index registers are also used. One of the indirect addressing methods is illustrated in examples in Chapter 7 and in Chapter 9.

Relative addressing

Relative addressing is one of the first addressing methods that was ever used, and it is not used for many commands nowadays. Relative addressing means that the operand of an instruction can be one or two bytes, and the address that is going to be used is found by adding this number (called the offset) to the 'current address', which is the number in the Program Counter. It's rather like the old-style Treasure Island maps which specify 'one step left, two forward, three right ...' and so on. You don't know where this will get you until you know where to start, but when relative addressing is used in a microprocessor, the starting place is usually the address in the PC. The 6502 uses relative addressing only for its BRANCH commands, and the offset is a single byte which is treated as a signed number. The use of a single byte signed number means that we can jump to a new address which is up to 127 steps forward or 128 steps back from the present one. These branches are the machine code equivalents of GOTO, but with the difference that they can be made to depend on a condition, like the accumulator containing zero. It's as if there were one single BASIC instruction which carried out the effect of:

```
IF A = Ø THEN GOTO...
```

We'll look at branch instructions in a lot more detail later.

The other registers

Of the other registers, the S register is an eight-bit register that we shall leave strictly alone for the moment. It is the type of register that is called a 'stack pointer', and it is used to locate bytes which the MPU has stored temporarily. If you interfere with what is stored in the S register, you may upset the operating system of the computer. The other register that is important to us is the Processor Status (P) register and we'll deal with it in more detail now. The Processor Status register, sometimes called the Flag register, isn't really a register like the others. You can't do anything with the bits in this register, and they don't even fit together as a number.

What the Process Status register is used for is as a sort of electronic note-pad. Seven of the bits in the register (there are eight of them altogether) are used to record what happened at the previous step of the program. If the previous step was a subtraction which left the A register storing zero, then one of the bits in the Process Status register will go from value Ø to value 1 to bring this to the attention of the MPU. If you add a number to the number in an accumulator, and the result consists of nine bits instead of eight (Fig. 4.2) then another of the bits in the Process Status register is 'set', meaning that it goes from Ø to 1. If the most significant bit in a register goes from Ø to 1 (which might mean a negative number), then another of the Process Status bits is set. Each bit, then, is used to keep a track of what has just happened. What makes this register important is that you can make branch commands depend on whether a Process Status bit is set (to 1) or reset (to Ø).

Number in accumulator	1Ø11Ø11Ø
Number added	11ØØØ1Ø1
Result	1Ø1111Ø11

This consists of nine bits, and the accumulator can hold only eight. The most significant bit is transferred to the carry flag of the status register.

Accumulator now holds Ø1111Ø11
Carry bit is set (equal to 1)

Fig. 4.2. Why the carry bit is needed.

Figure 4.3 shows how the bits of the Process Status register of the 6502 are arranged. Of these bits, Ø, 1 and 7 are the ones that we are most likely to use at the start of a machine code career. The use of the others is rather more specialised than we need at the moment. Bit Ø is the Carry Flag. This is set (to 1) if a piece of addition has resulted in a carry from the most significant bit of a register. If there is no carry,

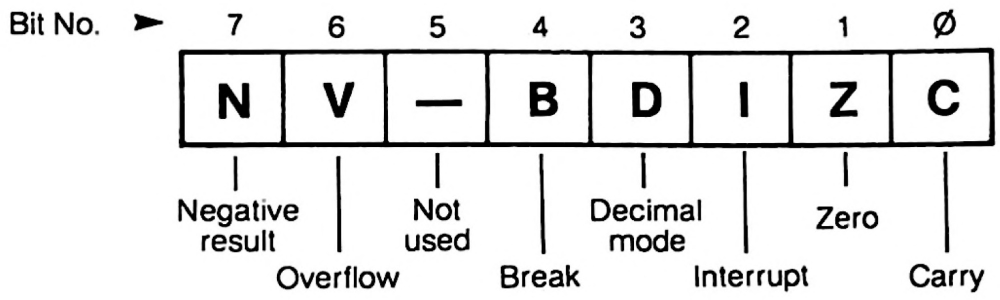


Fig. 4.3 The bits of the Processor Status register. Only three of these, N, Z, and C, are extensively used in most programs.

the bit remains reset. When a subtraction is being carried out (or a similar operation like comparison), then this bit will be used to indicate if a 'borrow' has been needed. It can for some purposes be used as a ninth bit for either accumulator, particularly for shift and rotate operations in which the bits in a byte are all shifted by one place (Fig. 4.4).

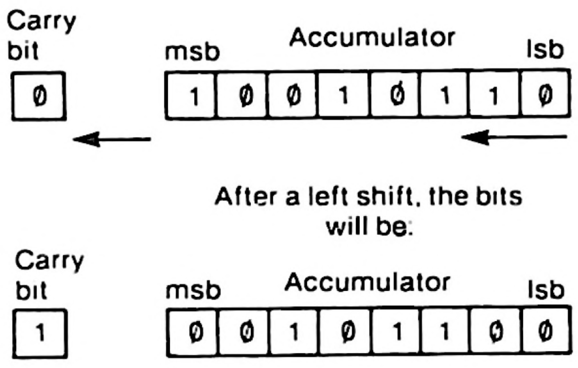


Fig. 4.4. Using the carry bit in a shift operation, in which all the bits of a byte are shifted one place to the left.

The Zero Flag is bit 1. It is set if the result of the previous operation was exactly zero, but will be reset (Ø) otherwise. It's a useful way of detecting equality of two bytes – subtract one from the other, and if the Zero Flag is set, then the two were equal. The Negative Flag is set if the number resulting in a register after an operation has its most significant bit equal to 1. This is the type of number that might be a negative number if we are working with signed numbers, or if we want to test that one number is bigger than another. This bit is therefore used extensively when we are working with signed numbers.

You can alter some of the bits of the Process Status register selectively – but that's not beginner's work! For the most part, we don't load anything into this register, or store its content. It is used almost exclusively as a way of keeping track of what has just been done, and that's how we shall illustrate its use in this book. Other dodges can wait until you're an expert.

Chapter Five
Register Actions

Accumulator actions

Since the accumulator is the main single-byte register, we must now list its actions and describe them in detail. Of all the accumulator actions, simple transfer of a byte is by far the most important. We don't, for example, carry out any form of arithmetic on ASCII code numbers, so that the main actions that we perform on these bytes are loading and storing. We load the accumulator with a byte copied from one memory address, and store it at another. Very few computer systems allow a byte to be moved directly from one address to another, so that the rather clumsy-looking method of loading from one address and storing to another is used almost exclusively.

The next most important group of actions is the arithmetic and logic group, which contains addition, subtraction, AND, and OR. We can add to this group the SHIFT and ROTATE actions which we looked at briefly in the previous chapter. The effects of the 6502's shift and rotate commands, with their assembly language mnemonics, is shown in Fig. 5.1. A shift always results in a register losing one of its stored bits, the one at the end which is shifted out Both types of shifts cause the register to gain a zero at the opposite end. The carry bit is used as a ninth bit of the accumulator in both of these shifts. The shift action can be carried out on either the 'A' register (the accumulator) or on a byte that is stored in the memory. The effect of a shift on a binary number stored in the register is to multiply the number by two if the shift is left, or to divide it by two if the shift is right (Fig. 5.2). A rotation, by contrast, always keeps the same bits stored in the register, but the positions of the bits are changed. The 6502 has two rotate commands, one for rotate right and the other for rotate left. Once again, they use the carry bit as the

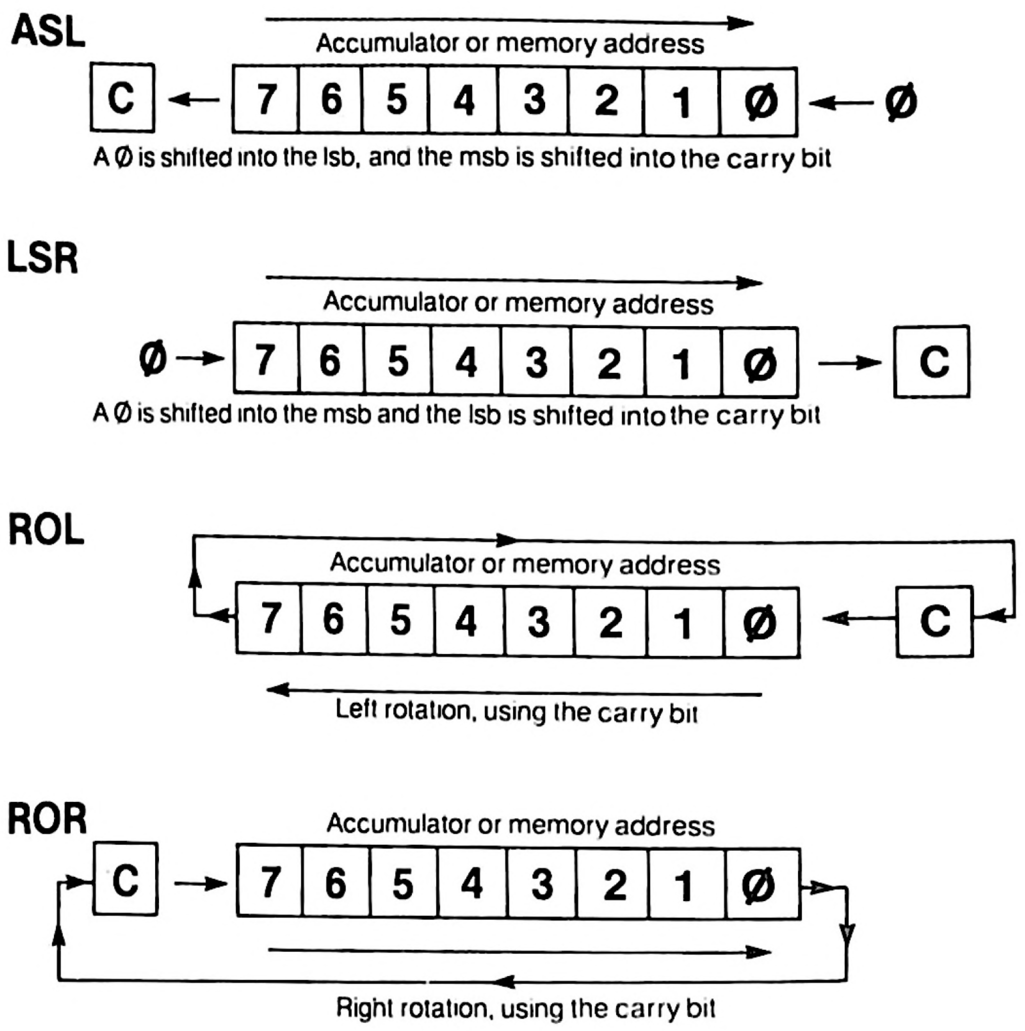

ASL
Accumulator or memory address
C
7 6 5 4 3 2 1 Ø
Ø
A Ø is shifted into the lsb, and the msb is shifted into the carry bit
LSR
Accumulator or memory address
Ø
7 6 5 4 3 2 1 Ø
C
A Ø is shifted into the msb and the lsb is shifted into the carry bit
ROL
Accumulator or memory address
7 6 5 4 3 2 1 Ø
C
Left rotation, using the carry bit
ROR
Accumulator or memory address
C
7 6 5 4 3 2 1 Ø
Right rotation, using the carry bit

Fig. 5.1. The 6502 shift and rotate instructions: ASL (Arithmetic Shift Left), LSR (Logic Shift Right), ROL (Rotate Left) and ROR (Rotate Right).

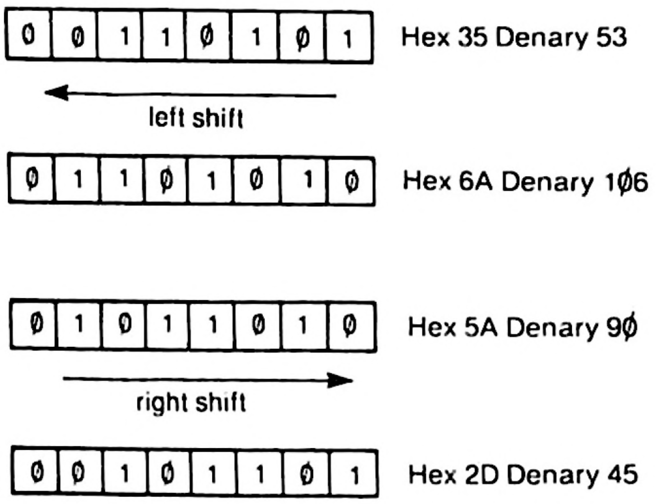


Fig. 5.2. The effect of a shift on a number.

ninth bit of the register. Either the accumulator can be used, or the action can be carried out on a byte stored in the memory.

The 6502, unlike most other microprocessor types, does *not* allow the accumulator to be used for increment or decrement actions. Increment means adding 1 and decrement means subtracting 1. These actions can be carried out on bytes stored in the memory, though, so that the 6502 programmer can increment or decrement bytes before or after use. This may mean that a byte has to be returned to memory to be incremented or decremented. It's just as easy, however, to use ADD immediate to perform an increment. An immediate subtract is not quite so simple, because the carry flag in the P register (the processor status register) must be set to 1 before a subtraction is carried out. If the carry flag is not set, the result of subtracting 1 will be to subtract 2! The accumulator, however, can be used for the important comparison commands.

The CMP (Compare) instruction is a particularly useful one. CMP is the mnemonic and it must use one of the standard memory addressing methods. The effect of CMP is to compare the byte that was copied from the memory with the byte that is already present and stored in accumulator A. Compare in this respect means that the byte copied from memory is subtracted from the byte in the accumulator. The difference between this instruction and a true subtraction is that the result is not stored anywhere! The result of the subtraction is used to set flags in the P register, but nothing else, and the byte in the accumulator is unchanged. For example, suppose that the accumulator contained the byte $4F, and we happen to have the same size of byte stored at address $327F. If we use the command:

```
CMP $327F
```

then the zero flag in the CC register will be set (to 1), but the byte in the accumulator will still be $4F, and the byte in the memory will also still be $4F. A subtraction would have left the content of the accumulator equal to zero.

Why should this be important? Well, suppose you want a program to do one thing if the Y key is pressed, and something different if the N key is pressed. If you arrange for the machine code program to store into the accumulator the ASCII code for the key that was pressed, then you can compare it. By comparing it with $4E (the ASCII code for 'N'), we can find if the N key was pressed. If it was, the zero flag will be set. If not, we can make another test. By comparing with $59, we can find if the Y key was pressed – once again, this would cause the zero flag to be set. If neither of these comparisons caused the zero flag to be set, we know that neither the Y nor the N key was pressed, and we can go back and try again. If it looks very much like the action that you can program using the GET$ loop in BASIC, you're right – it is.

Finally, we have the test-and-branch actions. These, as the name suggests, allow the flags in the P register to be tested, and will make the program branch to a new address if a flag was set. Which flag? That depends on which branch-and-test instruction you use, because there's a different one for each main flag, and for each state of a flag. For example, consider the two tests whose mnemonics are BEQ and BNE. BEQ means 'branch if equal to zero'. As this suggests, it will cause a branch if the result of a subtraction or comparison is zero. In other words, it causes a branch to take place if the zero flag is set. Its 'opposite number', BNE, means 'branch if not equal to zero'. It will cause a branch to take place if the zero flag is not set. There are, therefore, two branch instructions which test the zero flag, but in opposite ways. The same sort of thing goes for several of the other flags. There's also a different type of branch instruction, mnemonic JMP, which doesn't carry out any tests, like a GOTO with no IF preceding it.

The complete list of all the available branch instructions is shown in Fig. 5.3. Many of these are instructions that you'll probably never use, and the really important ones are the ones that use the zero, carry and negative flags. All of them, with the exception of JMP, use relative addressing. This means that one number byte must follow the code for the branch. This number is treated as a signed number (in other words, if it's more than $7F, denary 127, it's treated as negative) and is added to the address which is in the PC at the instant when the branch is carried out. The result of this addition is the address to which the branch goes, so the next instruction that is carried out will be the one at this new address. This type of branch, which uses a single byte 'displacement' number, permits a shift of up to 127 (denary) places forward, or 128 backward. That's because a single signed byte can't exceed these values.

Interacting with Oric/Atmos

The time has come now to start some practical machine code programming of your Oric/Atmos. This is not simply a matter of typing the assembly language lines as if they were lines of BASIC. Unless you happen to have an assembler program running, the

BCC Branch on carry clear: Jump to a new address if the carry flag is reset (at ∅).
BCS Branch on carry set: Jump to a new address if the carry flag is set (at 1).
BEQ Branch if equal to zero: Jump to a new address if the zero flag is set (at 1).
BNE Branch if not equal to zero: Jump to a new address if the zero flag is not set (zero flag at ∅).
BMI Branch on result minus: Jump to a new address if the result of the previous operation was a negative number (N flag set).
BPL Branch on result positive: Jump to a new address if the result of the previous operation was a positive number, or zero (N flag reset).
BVC Branch on overflow clear: Jump to a new address if the overflow flag is ∅.
BVS Branch if overflow set: Jump to a new address if the overflow flag is set. This will happen if the action of adding or subtracting causes the sign bit (the most significant bit) to change incorrectly.

Note: All of the above instructions use PC-relative addressing. The command byte has to be followed by a single byte, called the displacement, which is added to the address in the Program Counter register to obtain the new address.

JMP Jump to new address: Jump to the new address given by the two bytes that follow the JMP code.

Fig. 5.3. The complete list of 6502 branch instructions.

Oric/Atmos will simply give you the 'SYNTAX ERROR' message when you try to run these programs. Since we want to start on a small scale, we'll forget about assemblers at the moment, and assemble 'by hand'. This means that we find the machine code bytes that correspond to the assembly language instructions by looking them up in a table. We then have to convert the hex codes and data numbers into denary numbers. We then poke these numbers into the memory of the Oric/Atmos, place the address of the first byte into the PC of the 6502, and watch it all happen. It sounds simple, but there is quite a lot to think about, and a number of precautions to take. To start with, the Oric/Atmos uses quite a lot of its RAM, as we have seen, for its own purposes. If we simply POKE a number of bytes into the memory without heeding which part of memory we use, the chances are that we shall either replace bytes that the Oric/Atmos needs to use, or our program bytes will be replaced by the action of the Oric/Atmos. What we need is a piece of memory that is safely roped off for our use only.

This can be done by making use of the fact that the Oric/Atmos can reserve memory by using the BASIC instruction word HIMEM. The last byte of RAM that a program of your own can normally make use of is $98ØØ (denary 38912) for the 48K Oric or Atmos. For the 16K machines, the number is $18ØØ (denary 6144). Addresses right at the end of RAM memory like this are normally used for storing strings that are not already present lower in memory. We have already looked at this principle in Chapter 2. Now the Oric/Atmos is not programmed to stop at this number automatically. What is done is to store this 'end-of-RAM' number, as two bytes, in memory locations $A2 (low-byte) and $A3 (high-byte). These are zero page addresses, and in the course of operating a BASIC program, the computer will be continually testing that it does not try to use locations higher than the number stored in these addresses. You can find the HIMEM address in the usual way by peeking at the numbers in the usual way, but it's time now to make use of a little Oric/Atmos trick. If you use DEEK in place of PEEK, then you can carry out the peek of two addresses *and* the multiplication by 256 and addition that is needed to make up an address. Try, for example:

```
PRINT DEEK(#A2)
```

to find your HIMEM number. Easier, isn't it?

Suppose, then, that we alter this number. We don't have to alter the bytes directly, though we could do so by using either POKE or DOKE. DOKE will alter both bytes, so as to place an address into them as two bytes in low-higher order. If we type:

```
HIMEM 38399
```

for the 48K machine, then we will have reserved memory address 384ØØ to 38912 for our own uses. BASIC will not make use of any address higher than 38399, and therefore cannot interfere with our machine code programs. For a 16K machine, use HIMEM5631 instead of HIMEM38399.

The other problem, then, is how to place the starting address for your program into the program counter of the 6502. Fortunately, the designers of the Oric/Atmos have been kind to you. There is a BASIC command CALL which will do this for you. CALL has to be followed by a number, and this number will be placed into the PC. It will therefore be used as the address of the first byte of your program. Incidentally, I've taken this as meaning 'starting byte'. It's possible to write programs in which the first few bytes are data, so that the program starts at, say, the tenth byte. This creates no problems; you simply use the address of the starting byte as the number for CALL.

Lastly, for the moment at least, you have to ensure that your machine code program will stop in an orderly way. Nothing that we have done so far will indicate to the 6502 of the Oric/Atmos where your program ends. As a result, the 6502 could continue to read bytes after the end of your program, until it encounters some byte which causes a 'crash'. This might, for example, be a byte which causes an endless loop. Some programmers doubt if there are any bytes which do *not* cause an endless loop in these circumstances! To return correctly to the operating system of the Oric/Atmos, you need to end each machine code program with a 'return from subroutine' instruction, whose mnemonic is RTS and whose code is $6∅.

There's another headache that we don't have to worry about at the moment. When you run a machine code program along with a BASIC program in your Oric/Atmos, you are using the same 6502 microprocessor for both jobs. It can't cope with both at the same time, so it runs one, then the other. If you make use of the 6502 registers in your machine code program, as you are bound to do, then you have to be quite certain that you are not destroying information that the BASIC program needs. For example, if at the instant when your machine code program started, the registers of the 6502 contained the address of a reserved word in the ROM, then it will need this address in these registers when your machine code program ends. When a machine code program is called into action by using the CALL command, this is taken care of automatically. The contents of the registers of the 6502 are placed into a part of the RAM memory which is called 'the stack'. This, incidentally, is another good reason for being careful as to where you place your machine code in the memory. If you wipe out the stack, the Oric/Atmos will quite certainly not like it! The stack is located in the range of memory between addresses 256 and 511. When the RTS instruction is encountered at the end of your machine code, the bytes that have been stored in the 'stack' are replaced into their registers, and normal action resumes. If you call a machine code program into action by any other method, not using CALL, you may have to attend to this salvage operation for yourself as part of your machine code program. This involves using the PUSH and PULL commands – but more of that later.

Practical programs at last!

With all of these preliminaries out of the way, we can at last start on some programs which are very simple, but which are intended to get you familiar with the way in which programs are placed into the memory of the Oric/Atmos. You will also get some experience in the use of assembly language and machine code, and with how a machine code program can be run.

We'll start with the simplest possible example – a program which just places a byte into the memory. In assembly language, it reads:

```
ORG $96ØØ; start placing bytes here
LDA #$55; place hex 55 in accumulator
STA $9614; store them at $9614
RTS; go back to BASIC
```

The first line contains a mnemonic, ORG, which you haven't seen before. It isn't part of the instructions of the 6502, but it is an instruction to the assembler, which in this case is you! ORG is short for origin, and it's a reminder that this is the first address that will be used for your program. We've chosen to use an address which leaves space for longer programs than we shall be writing in the course of this book, and we could have chosen a higher number. It will do as well as any other, however, and it leaves plenty of room for longer programs. When you program using an assembler, this line can be typed and the assembler will then automatically place the bytes of the program in the memory starting at this address. As it is, with assembly being done 'by hand' it simply acts as a reminder of what addresses to use. Note the comments which follow the semicolons. The semicolon in assembly language is used in the same way as a REM in BASIC. Whatever follows the semicolon is just a comment which the assembler ignores, but which the programmer may find useful.

Now we need to look at what the program is doing. The first real instruction is to load the number $55 into the 'A' accumulator. This uses immediate addressing, so the number $55 will have to be placed immediately following the instruction. The hashmark, #, is used in assembly language to indicate that immediate loading is to be used. The next line commands the byte in the accumulator (now $55) to be stored at address $9614. In denary, this is 3842Ø. It's an address well above the ones that we shall use for the program. Obviously, we wouldn't want to use an address which was also going to be used by the program. This instruction uses absolute addressing. Finally, the program ends with the RTS instruction, essential for ensuring that Oric/Atmos life continues normally after our program ends.

The next step in programming is to write down the codes in hex. Each code has to be looked up, taking care to select the correct code for the addressing method. The code for LDA immediate is $A9, so this is the first byte of the program which will be stored at address $96ØØ, denary 384ØØ. We can start a table of address and data numbers with this entry:

```
384ØØ   $A9
```

and then move on. The byte that we want to load is $55, and this has to be put into the next memory address, because this is how immediate addressing works. The table now looks like this:

```
384ØØ   $A9
384Ø1   $55
```

The next byte we need is the instruction byte for STA, with absolute addressing. This byte is $8D, and it has to be followed by the two bytes of the address at which we want the bytes stored. The address 3842Ø translates into hex as $9614, so we can use the bytes $14 and $96 following the STA instruction. Remember that these bytes have to be in low-high order. The last code has to be the RTS code of $6Ø, so that the table now looks as in Fig. 5.4. It uses addresses 384ØØ to

Address (Denary)	Byte (Hex)
384ØØ	$A9
384Ø1	$55
384Ø2	$8D
384Ø3	$14
384Ø4	$96
384Ø5	$6Ø

Fig. 5.4. The coded program, using denary addresses and hex bytes of data.

384Ø5, six bytes in all, and will place a byte into 3842Ø, using denary numbers. Now we have to put it into memory and make it work!

This requires a BASIC program which will reserve the memory, and poke the bytes in one by one. Since the Oric/Atmos has its own hex-denary conversions, we can store the bytes in hex in a DATA line, and just poke them into place. There are two ways that this can be done, using POKE or DOKE. The POKE program is shown in

(a)
```
10 HIMEM38399:A=38400
20 FOR N=0 TO 5:READ D$
30 POKE A+N,VAL("#"+D$)
40 NEXT:CALL A
50 DATA A9,55,8D,14,96,60
```

(b)
```
10 HIMEM 38399:A=38400
20 FOR N=0TO5STEP2:READ D$
30 DOKE(A+N),VAL("#"+D$)
40 NEXT:CALL A
50 DATA 55A9,148D,6096
```

Fig. 5.5. The BASIC program which puts the bytes into place: (a) using POKE (b) using DOKE. Note how the order of pairs of bytes has to be reversed to use DOKE.

Fig. 5.5(a). By using HIMEM38399 we ensure that all memory addresses above 38388 are left unused by the Oric/Atmos. We declare the variable A as 384ØØ, so that we can make use of this in the POKE commands. Lines 2Ø to 4Ø then poke data numbers into addresses that start at 384ØØ. All of the codes have to be put into denary form for use with POKE, and this is done by using VAI ("#"+D$) in the program. It's better in many ways to work in hex as much as possible, and convert to denary only if you must. Since you have to make use of hex when you graduate to an assembler it's as well to start getting familiar with the principles of working in hex now.

The last program line, line 4Ø, contains CALL A. This is the BASIC instruction which will cause your machine code program to run, with the start address specified as variable A. Line 5Ø then contains the six bytes of data that we have worked out. When you RUN this, there's no obvious effect. That's because you can't see what's in address $9614. If you use:

?PEEK(#9614)

then you should find the value of 85, which is the denary version of $55, the number that the program put there. Now try this: type POKE#9614,255, press RETURN, and then delete the CALL instruction from your program. This is at the end of line 4Ø. RUN the program again, and use ?PEEK(#9614) to find what's there. It should be 255. Now type CALL#96ØØ and press RETURN. Using ?PEEK (#9614) should now give you 85 again. This is because poking the bytes of the program into memory won't make the program run, only CALL does this. You can therefore poke values into memory early in a BASIC program, and then make use of them later with a CALL wherever you like.

Now this program isn't an ambitious piece of work, it does no more than POKE#9614,#55 would do in BASIC, but it's a start. The main thing at this point is to get used to the way in which machine code operates, and how you place it into memory and run it. Another point, incidentally, is that the machine code is safe in memory. If you type NEW (RETURN), the BASIC program will be cleared out, but your machine code remains. If you POKE#9614,255 now, and test with ?PEEK(#9614), you will find that this address can still be changed by using CALL#96ØØ. These bytes will remain there until you make an effort to change them, or you switch off. You can preserve the machine code program on tape if you like, and this is a technique that we'll look at later. One step at a time, if you please! Another thing that we haven't space to cover is the alternative method of calling up a program, using the USR command.

Before we leave this example, however, take a look at Fig. 5.5(b). This places the same bytes into the memory, but uses DOKE in place of POKE. DOKE has two effects. One is that it places two bytes into consecutive addresses, starting with the address that follows the DOKE command. The other effect is that the bytes are placed in the order that you would use for an address – low, then high. For example, if you use DOKE #96ØØ,#1ØF4, then the byte $F4 is placed into address $96ØØ, and the byte $1Ø is placed in address $96Ø1. If we use DOKE, then, we have to group bytes into twos in the DATA line, and also alter the order of each pair. It's something for later, when you are really accustomed to the 6502 way of doing things.

Now let's try something a lot more ambitious in terms of our use

Assembly language	Code
LDX #$ØØ	A2 ØØ
LDA $962Ø,X	BD 2Ø 96
ASL A	ØA
INX	E8
STA $962Ø,X	9D 2Ø 96
RTS	6Ø

Fig. 5.6. The assembly language program for 'multiply by two'. The listing shows the assembly language on the left, and the hex codes on the right.

of machine code – though the example is simple enough. Figure 5.6 shows the assembly language version of the program. What we are going to do is to load a byte into the accumulator, shift it one place left, and then put it into memory at an address one step higher than the address from which we took it. This looks like an open-and-shut case for indexed addressing, so we shall start by placing a zero into the X register. This is the LDX #$ØØ step. As before, the # in assembly language means immediate addressing – don't mix it up with the way that the Oric/Atmos uses # for hex to denary conversion. The next line, LDA $962Ø,X means that the accumulator is to be loaded from the address $962Ø, plus the number in the X register. This makes the load from $962Ø (38432 denary). The third step is ASL A, arithmetic left shift of the byte in the accumulator, so that the bits of the byte are shifted left. Fourthly, we increment the number in the X register, using INX. This makes the Ø into 1, and we now store the byte in the accumulator at address $9621 by using STA 962Ø,X. This time, because 1 has been added to the number in the X register, the byte is stored at address $9621. We end, as always, with the RTS instruction.

Now we can put this into code form. It's just as easy as before, despite the use of indexing. The LDX instruction needs the immediate loading code of $A2, and this has to be followed by the byte of data, $ØØ. The LDA with indexed addressing is coded as $BD, and it has to be followed by the two bytes of the address $9FC4, in their usual low-high order. The ASL byte is ØA, and then we perform the incrementing of X with INX, code $E8. We then store the byte that is in the accumulator back into memory, using STA $962Ø,X. The STA X-indexed code is $9D, and this is followed by the now familiar address bytes. Finally, $6Ø is the RTS command.

Now we have to code this in BASIC. If we choose a small number to place into $962Ø, the effect of the left shift will be to double the

```
 10 HIMEM 38399:A=38400
 20 FORN=0 TO 10:READ D$
 30 POKE A+N,VAL("#"+D$)
 40 NEXT:POKE#9620,5:CALLA
 50 DATAA2,0,BD,20,96,0A,E8,9D,20,96,60
 60 PRINT"TWICE "PEEK(#9620)" IS "PEEK(#
9621)
```

Fig. 5.7. The BASIC program which pokes the bytes into place and then makes use of the machine code program.

number, so we can use this to obtain a bit of arithmetic wizardry. The BASIC program is shown in Fig. 5.7. We start, as usual, by clearing memory space. You needn't worry if you have had another program in this part of the memory before. The new program will replace it completely, and provided that your program ends correctly with the RTS instruction byte, the old program bytes cannot interfere with the new ones. The values are poked into places in the usual way in lines 2Ø to 3Ø. In line 4Ø, however, we place a number, 5 denary, into the address $962Ø. Now this is the address which will be used by the program, and the byte which is 5 in denary is, in binary form, ØØØØØ1Ø1. This byte will be placed in the address $962Ø. In the third part of line 4Ø, CALLA will carry out the machine code program, which should left-shift this byte, making it ØØØØ1Ø1Ø. In denary, this is 1Ø, twice 5. Line 6Ø prints this result, and line 5Ø contains the data bytes.

It's simple enough, but if you knew nothing about machine code, you would wonder how on earth the number became multiplied by two. Once again, the program does nothing that could not be done more easily and as quickly by using BASIC. The important thing, from our point of view, is that you have now used indexed addressing, and a shift instruction, as well as getting more experience in putting a machine code program into your Oric/Atmos by the hardest method of all. If, incidentally, you have made any mistakes, particularly with DATA, then it's likely that the Oric/Atmos will go into a trance and refuse to do anything. When you have typed in a BASIC program like this which pokes bytes into the memory, always record the BASIC program before you RUN it. This way, if the effect of an incorrect byte is to zonk out half the RAM, you can switch off, then on again, and reload your program, If you didn't record it, then you'll have to type it all over again. That's hard work, and life is hard enough as it is.

Another final point concerns the number that you poke into $962Ø. If this is a small number, then the program works. You can't, however, poke a number greater than 255 into any single memory position. In addition, if the number that you place there amounts to more than 254 when it is multiplied by two, then the results of this program will appear very strange! That's because a number greater than 255 cannot be stored in one memory byte. The routines that multiply numbers in your Oric/Atmos are a lot more sophisticated than this simple example!

Chapter Six
Taking a Bigger Byte

The simple programs that we looked at in Chapter 5 don't do much, though they are useful as practice in the way that machine code programs are written. Practising assembly language writing and its entry into the Oric/Atmos is essential at this stage, because you can more easily find out whether you are making a mistake when the programs are so simple. It's not so easy to pick up a mistake in a long machine code program, particularly when you are still struggling to learn the language!

Most beginners' difficulties arise, oddly enough, because machine code is so simple, rather than because it is difficult. Because machine code is so simple, you need a large number of instruction steps to achieve anything useful, and when a program contains a large number of instruction steps, it's more difficult to plan. The most difficult part of that planning is breaking down what you want to do into a set of steps that can be tackled by assembly language instructions. For this part of the planning, flowcharts are by far the most useful method of finding your way around. I never think that flowcharts are ideally suited for planning BASIC programs, but they really come into their own for planning machine code.

Flowcharts

Flowcharts are to programs as block diagrams are to hardware – they show what is to be done (or attempted) without going into any more detail than is needed. A flowchart consists of a set of shapes, with each shape being the symbol for some type of action. Figure 6.1 shows some of the most important flowchart shapes for our purpose (taken from the standard set of flowchart shapes). These are the *terminator* (start or stop), the *input/output*, the *process* (or action) and the *decision steps*. Inside or beside the shapes, we can write brief

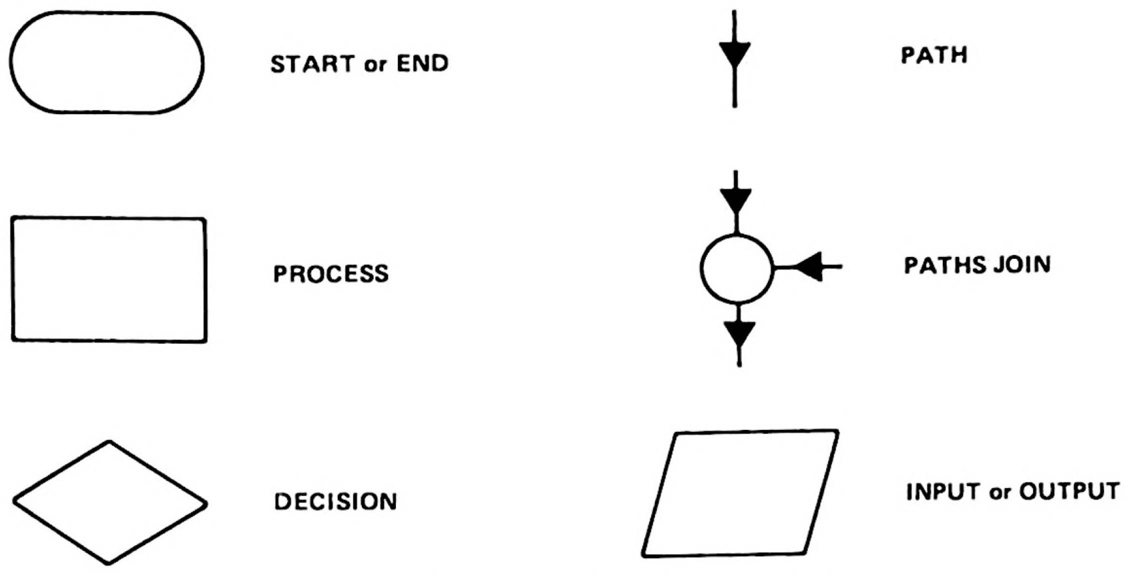


Fig. 6.1. The main flowchart shapes.

notes of the action that we want, but once again without details. The purpose of these symbols is to give you a visual picture of what your program is intended to do. it need not show details – not at first, anyway – but it must show methods.

Consider, for example, the flowchart in Fig. 6.2. This is the flowchart for a program which will read a byte from one address and write it to another. It's a very simple piece of work, and that's why I've used it as an example. The flowchart shows the steps, some names, and the direction of each step – but that's all. There's no mention of accumulators or of addressing methods. The flowchart is about *what* you want to do, and the order in which things are done, not about the details of how you do it. The name 'flowchart' should also remind you of the very important point about direction of

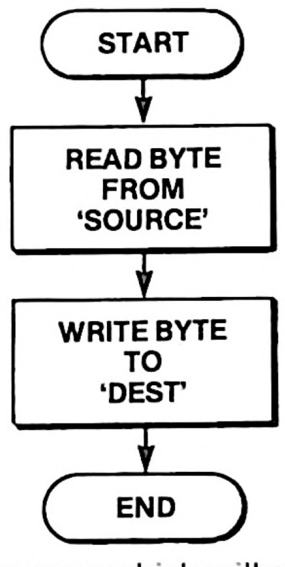


Fig. 6.2. A flowchart for a program which will read a byte from one address and store it at another.

'flow', showing how one step comes after another. What makes flowcharts important for machine code programmers is that it's easy to forget what you want to do! Machine code is very much about fine details, and it's easy to get so involved in the details that you forget what you are supposed to be doing. It's rather like the guy who is doing a crossword, looks up a word in a dictionary, and gets so interested that he carries on reading the dictionary. The usefulness of the flowchart starts to come into its own when we start looking at loops — right now.

The Power of the loop

We can, if we like, divide programs into two kinds, 'linear' and 'looping'. A linear program starts, carries out each line of the program, then stops. A looping program, by contrast, will repeat some lines many times over until some condition is satisfied. As you know from programming in BASIC, the ability of the computer to carry out looping programs is one of its most important features.

Very few machine code programs are linear, because the speed advantages of machine code are noticeable only when you use loops. In this chapter, then, we are going to look at several different types of loops, and how they are programmed. On the way, we shall learn a lot more about the action of the Oric/Atmos system. We'll start with some simple 'shifting loops'.

A 'shifting loop' is one that shifts bytes from one set of memory addresses to another. Why should we want to do this? Well, as you will have seen earlier, it's one way of shifting the position of an object on the screen. If we have an object which is represented by a set of bytes stored in screen memory, then shifting these bytes corresponds to shifting the object on the screen. We're not quite ready for that sort of programming yet, so we'll start small. The first problem, then, will be to move a set of bytes (a short BASIC program) from one set of addresses to another.

Now, once again, we're not doing anything that could not be done (more easily) in BASIC. It's the *method* that's important, because the methods that you learn on simple examples are a guide to the methods that you have to use for more complicated examples where BASIC is not so useful. We'll start with a flowchart, which is in Fig. 6.3. The flowchart shows that we pick two starting numbers. One of these, labelled START, is the address at which free memory starts when you first switch on, $Ø5ØØ. The other address, NEW, is a different value of $2ØØØ. What this flowchart is designed to do, then,

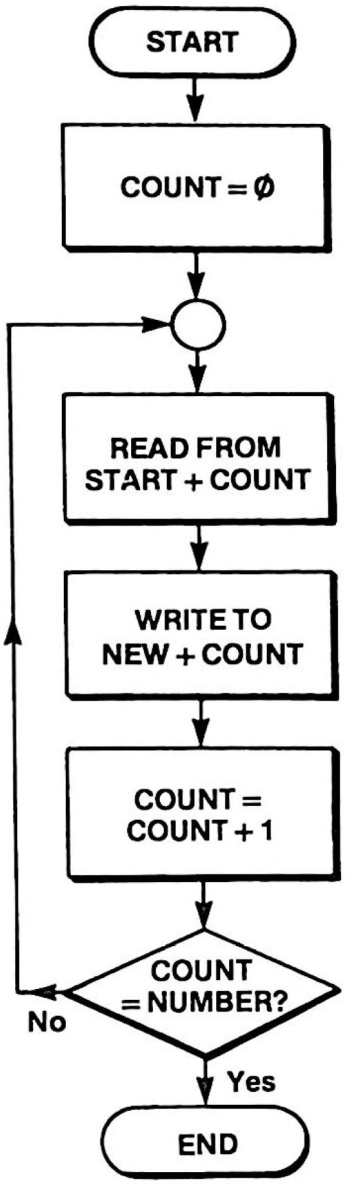


Fig. 6.3. A flowchart for copying a set of bytes to a different set of addresses.

is to copy a byte from $Ø5ØØ to $2ØØØ, then from $Ø5Ø1 to $2ØØ1, from $Ø5Ø2 to $2ØØ2, and so on. The number of bytes has to be fixed. In this example, I have taken the figure of 2ØØ (denary), $C8 in hex, which is enough to copy the example program itself.

The flowchart then shows what is to be done. A counter is set up, starting at zero. A byte is copied, and the counter is incremented. The addresses are also incremented. The counter is tested in a decision step. A decision step in a flowchart must have only two possible answers, YES or NO. This test is to see if the count has reached the value of 'NUMBER' (set at 2ØØ). If it has not, the loop is repeated, and bytes are copied until the count reaches 2ØØ. Now we have to find out how to make all this work using machine code. To start with, we shall plan to use indexed addressing. This neatly ties up the incrementing of the counter, and both of the addresses. If we use X-indexed addressing, then the count number can be held in register X. Each time we increment X, then an indexed load from $ØEØØ will, in fact, be from the address $ØEØØ+X, and each store to $23ØØ will be to $23ØØ+X. We can test for the end of the program by testing the value of the byte in register X.

(a)

	Assembly language	Code
	ORG $96ØØ	———
	LDX #$ØØ	A2 ØØ
LOOP:	LDA $Ø5ØØ,X	BD ØØ Ø5
	STA $2ØØØ,X	9D ØØ 2Ø
	INX	E8
	CPX #2ØØ	EØ C8
	BNE LOOP	DØ F5
	RTS	6Ø

(b)

```
10 HIMEM#95FF:A=#9600
20 FORN=0TO13:READ D$
30 POKE A+N,VAL("#"+D$):NEXT
40 CALL A
100 DATA A2,0,BD,0,5,9D,0,20,E8,E0,C8,D
0,F5,60
```

Fig. 6.4. The copy program (a) assembly language, (b) the BASIC poke program.

Figure 6.4(a) shows an assembly language program which will carry out the action of the flowchart. Don't run away with the idea that this is the only program which will carry out these actions. You could devise several programs that would carry out (or 'implement') the flowchart actions. Of these, one might be much faster in operation than the others. Another might require fewer bytes of code than any of the others. The testing for a value in X, for example, would be much easier if you started with a number in the X register and decremented the register. There's seldom any single 'perfect' solution to a problem, so don't take the sample programs in this book as being anything more than aids to learning. The perfect program, as far as you are concerned, is one that does exactly what you want it to do.

Now examine the program in detail. It starts as usual with a reminder of the address at which the code is to be located, $96ØØ.

The next step is to load zero into the X register, using an immediate loading. The third line, however, introduces a new idea, a *label.* A label is a word that represents a byte or an address, just as a variable name represents a number. A label is attached to a place in an assembly language program, by using the label name ahead of the opcode, separated by a colon. Its meaning is, quite simply, 'this address'. The label LOOP in this line means the address that is stored in the Program Counter at the instant when this line is carried out. We know what this address must be, because if the LDX instruction byte is a $96ØØ, the LOOP label must be at address $96Ø2. In assembly language, however, it's a lot more convenient to mark the instruction with a label than to have to work out addresses. If you use an assembler, the action of typing a label will ensure that the address is memorised and used.

Now for the action. The next line gets a byte from the starting address. With the X-register storing zero, the load will be from address $Ø5ØØ. This byte is then stored by the next command at address $2ØØØ, again because the value of the byte in X is zero. The X register is then incremented. On the first pass through this loop, this makes the byte in X equal to 1. The CPX #2ØØ step then tests this number, to see if it is equal to 2ØØ. If it is not equal, the BNE (branch if not equal to zero) command causes the program to loop back. By using the label word 'LOOP' following BNE, you ensure that the program shows where you must loop back to. When, eventually, the number stored in the X register reaches 2ØØ, the BNE test fails, there is no loop back, and the RTS is carried out. The effect of the complete program is to copy a block of 200 bytes which start at $Ø5ØØ, to another block of addresses starting at $2ØØØ.

Now try it out, using the BASIC POKE program in Fig. 6.4(b). Note that what follows the BNE instruction byte of $DØ is a 'displacement' byte of $F5. This is the negative number which, when it is added to the address number in the program counter, will get back to the address number of the LDA START,X command. Because the PC register automatically increments after each byte is read, this negative number is one step more negative than you might expect from subtracting addresses. Figure 6.5 shows how these displacements have to be calculated. If you use an assembler, then the system of using label words ensures automatically that the displacement bytes are always correctly calculated – if they can be. The catch is that you can't use relative addressing for a displacement of more than 127 forward or 128 back. If you try, the assembler will stop and issue an error message. Without an assembler, you have to check and double-check each displacement byte very carefully. If a program goes berserk, it's almost always because of a faulty displacement byte.

When the program of Fig. 6.4(b) runs, you don't see much

Destination: the address that you want to jump to. This will have a label name ahead of it in the assembly language line.
Source: the address that you want to jump from, the address of the instruction which has the label name *following* it.
Displacement: the byte equal to destination minus source minus 2, put into hex form.

Example:

```
9620    173
9621    0
9622    2
9623    240
9624    displacement byte
```

The 'source' address is 9623, where the BEQ byte is placed. The 'destination' address is 9620, the LDA command.
The procedure is:
Destination address — source address = 9620 − 9623 = −3
Now subtract another 2, so that −3−2=−5.
In denary, the equivalent byte for −5 is 256−5 = 251. This must be the byte at address 9624. In hex, this displacement is $FB.

How to check displacement bytes:
Positive (forward) jump. Count the number of bytes from the next instruction to the byte just before the destination. This should be equal to the displacement byte.

Negative (backward) jump. Count the displacement byte itself as FF. Count back, FE, FD, FC ... as you go each step back. The number you have reached when you get to the correct label name is the value of the displacement byte.

Fig. 6.5. Finding the size of a displacement byte, and checking the displacement.

evidence of it. If, however, you carry out DOKE#A9,#2001, which shifts the start of BASIC to address 2001, you'll find when you LIST that you have a program at this address. You can go back to the original program by using DOKE#A9,#501. If you NEW this program, you will find that the copy still exists at #2001 (use DOKE#A9,#2001 again to check). You cannot RUN the copy.

however, because the addresses for the line numbers following line 1∅ are wrong.

Just at this stage, incidentally, it's interesting to look at what NEW does. If you read the first ten bytes of a BASIC program before you NEW it, and then again after, you'll find that NEW affects only the first two. These first two are set to ∅, which is the way that the Oric/Atmos marks the end of a program. To restore a program after NEW, then, you need only restore these first two numbers. We'll look at this point again later.

Count your loops

It's time now to take a look at another type of loop, called a *counting loop*. The importance of this one is twofold – it's the way that we program a time-delay in machine code, and it also gives me an excellent opportunity to demonstrate just how fast machine code can be.

The type of loop that you use most in BASIC is the FOR...NEXT loop. This uses a 'counter' variable to keep a score of how many times you have used the loop, and compares the value of the counter with the limit number that you have set each time the loop returns. Now the action of a FOR...NEXT loop can be simulated in BASIC without using FOR or NEXT, and one method is shown in Fig. 6.6.

```
10 C=0:ND=10
20 PRINT"ACTION ";C
30 C=C+1
40 IF C<=ND THEN 20
50 PRINT "FINISHED"
```

Fig. 6.6. A simulated FOR...NEXT loop. This uses methods that are very similar to the methods we have to use in machine code.

The count number is C, and its limit is ND. At the end of the program, the value of C will be 11, just like the value of the counter in a FOR N= 1 TO 1∅ type of loop. The next thing, then, is to take a look at the flowchart for this type of program, and that's shown in Fig. 6.7.

This method of forming a counting loop is the one that we use in machine code. We can write some assembly language that will do the same job – but, as usual, we have to give a lot more thought to how the task will be done. For one thing, we don't have variable names in machine code. We have to decide where we shall store a number, and in what register we shall carry out the task of decrementing it. The decision step is easier – we can use a BNE test this time to keep the program looping back until the content of the register that we have tested is zero. In case you're wondering how we specify which register we're testing, the answer is that it's always the one that we used just before the BNE (or any other) test.

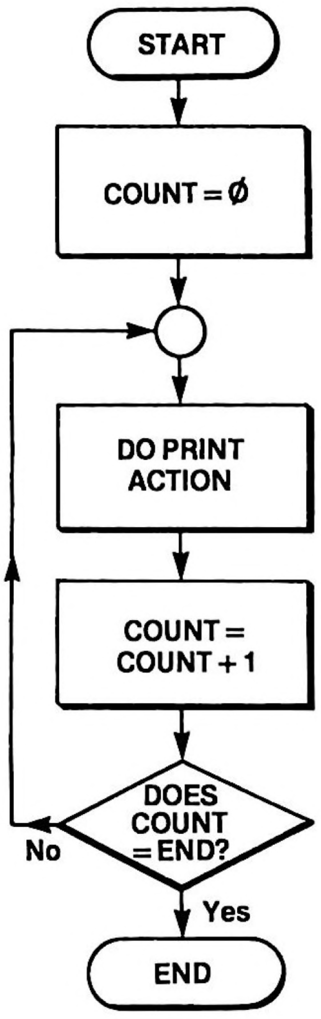


Fig. 6.7. A flowchart for a counting loop.

(a)

	Assembly language	Code
	ORG $96ØØ	———
	LDX #$FF	A2 FF
LOOP:	DEX	CA
	BNE LOOP	DØ FD
	RTS	6Ø

(b)
```
10 HIMEM#95FF:A=#9600
20 FORN=0TO5:READ D$
30 POKEA+N,VAL("#"+D$):NEXT
40 PRINT"START":CALLA:PRINT"STOP"
100 DATAA2,FF,CA,D0,FD,60
```

Fig. 6.8. (a) The assembly language and (b) the BASIC poke program for a counting loop.

Figure 6.8(a) shows what we end up with as an assembly language program. The register that we use is the X register, rather than the accumulator. This is because the X-register (along with the Y one) is better adapted for counting operations. What makes it better adapted is the presence of increment and decrement commands. INX and DEX will, respectively, increment or decrement the X register. The commands INY and DEY will perform the same operations on the Y register. There is no corresponding pair of commands for the accumulator, and if we want to carry out counting operations using the accumulator, we have to go about them in a slightly different way. In this example, then, the X-register is loaded with $FF, which is 255 in denary. This is the largest number that we can load into an eight-bit register. Having loaded the accumulator, we then decrement it, and mark this address as 'LOOP', the place we want to return to if the register content is not zero. The test is carried out by BNE, branch if not equal to zero, because we want the program to repeat the decrementing action until the contents of the X register reach zero.

In the BASIC poke version in Fig. 6.8(b), the machine code is called in line 4∅. Now when you run this one, you will not see much of a time delay between the printing of 'START' and the printing of 'STOP'. This isn't because nothing has happened, it is because the machine code count-down is so fast! Figure 6.9 shows a BASIC

```
10 C=0:ND=10
20 PRINT"ACTION ";C
30 C=C+1
40 IF C<=ND THEN 20
50 PRINT"FINISHED"
```

Fig. 6.9. A BASIC version of the machine code counting program.

version of this count-down, using the same number, and you can see that there is a noticeable pause. The difference does not reflect the comparative speeds, however, because quite a lot of time is spent in the printing actions of the BASIC part of each program. To see just how great the advantage of machine code can be in terms of speed, we need to work with much larger numbers. Now there are several ways of doing this, but one which we can look at right now involves two loops.

You have probably met nested loops in BASIC. The principle is that there is an inner loop and an outer loop. On each pass of the outer loop, the whole of the inner loop is carried out. This allows us to create much longer time delays, by doing one count inside another. Suppose we had, in BASIC, the lines:

```
1Ø X=255:PRINT"START"
2Ø X=X-1
3Ø Y=255
4Ø Y=Y-1
5Ø IF Y<>Ø THEN 4Ø
6Ø IF X<>Ø THEN 2Ø
7Ø PRINT"STOP"
```

then these would carry out a count-down of Y from 255 to Ø each time the value of X was decremented. Try this one – and time it. You won't need a stop-watch – anything with a minute hand will do!

For a contrast, let's see how the same numbers could be dealt with in a machine code count-down. Figure 6.10(a) shows the assembly

(a)

```
          Assembly language     Code

          ORG $96ØØ             ———
          LDX #$FF              A2 FF
LOOP2:    LDY #$FF              AØ FF
LOOP1:    DEY                   88
          BNE LOOP1             DØ FD
          DEX                   CA
          BNE LOOP2             DØ F8
          RTS                   6Ø
```

(b)

```
10 HIMEM#95FF:A=#9600
20 FOR N=0TO10:READ D$
30 POKE A+N,VAL("#"+D$):NEXT
40 PRINT"START":CALL A:PRINT"STOP"
100 DATA A2,FF,A0,FF,88,D0,FD,CA,D0,F8,
60
```

Fig. 6.10. A nested loop counting program. This takes considerably longer to execute.

language version. The X-register is loaded with $FF, and the Y-register also with $FF. This second instruction is labelled 'LOOP2'. Then comes DEY, so that the Y-register is decremented, and this is labelled 'LOOP1'. The BNE test then returns to this LOOP1 point (the inner loop) until the Y register has reached ∅. After that, the X register is decremented, and then tested. Note that the order is not quite the same as in the BASIC version. In a machine code decrement and test action, you must have the decrement done just before the test, otherwise the register that is tested may not be the correct one. If the X register has not reached zero, the program loops back, this time to Loop2, to fill up the Y-register again and perform the 'inner loop' yet again.

When you try this, it's still almost too fast to follow! It's a good illustration of the speed advantage of machine code as compared to BASIC. Just compare the speeds of the two counts. The machine code causes a slight delay. The BASIC version ... well, it gives you time for a coffee. For speed – use machine code.

Accumulator INC and DEC

I pointed out earlier that there are no INC and DEC commands for the accumulator. This does not, however, mean that we can't use the accumulator for counting operations, just that it's not so well equipped as the X- and Y-registers. Supposing we have to carry out a count in the accumulator, however, we can make use of SBC #1, meaning subtract 1 from the content of the accumulator. This would lead us to a count-down program of the sort that is shown in Fig. 6.11. We start by using SEC, the command which sets the carry bit.

(a)

	Assembly language	Code
	ORG $96∅∅	----
	SEC	38
	LDA #$FF	A9 FF
LOOP:	SBC #1	E9 ∅1
	BNE LOOP	D∅ FC
	RTS	6∅

(b)
```
10 HIMEM#95FF:A=#9600
20 FOR N=0TO7:READ D$
30 POKE A+N,VAL("#"+D$):NEXT
40 PRINT"START":CALL A:PRINT"STOP"
100 DATA38,A9,FF,E9,1,D0,FC,60
```

Fig. 6.11. (a) The assembly language version of the program and (b) the BASIC poke program.

This is needed, because if the carry bit happens to be clear, then the first SBC action will subtract 2 rather than 1. All the remaining SBC actions will be normal, and it would not make much difference to a time delay like this to have an extra decrement. In some kinds of decrement action, however, like counting bytes, the subtraction of 2 instead of 1 could be a disaster. For this reason, it's a good habit to set the carry bit before you carry out a subtraction of this sort. The SBC #1 is the decrementing step, and the loop runs pretty much like the X register loop that we looked at earlier.

The 6502 does not confine you to decrementing numbers that are stored in the registers, however. The DEC action can be applied to any address in memory, so you can write count-down programs that use as many bytes as you like. The most convenient addresses to use for storing counting bytes are the zero page addresses (from Ø to 255), but when your 6502 is installed in an Oric/Atmos, you have to be careful! This is because the Oric/Atmos uses several of the zero page addresses for its own purposes, and placing bytes in some of

(a)

	Assembly language	Code
	ORG$/96ØØ	———
	CLC	38
	LDA #$FF	A9 FF
	STA $7Ø	85 7Ø
LOOP1:	STA $71	85 71
LOOP2:	STA $72	85 72
LOOP3:	DEC $72	C6 72
	BNE LOOP3	DØ FC
	DEC $71	C6 71
	BNE LOOP2	DØ F6
	DEC $7Ø	C6 7Ø
	BNE LOOP1	DØ FØ
	RTS	6Ø

(b)
```
 10 HIMEM#95FF:A=#9600
 20 FORN=0TO21:READ D$
 30 POKEA+N,VAL("#"+D$):NEXT
 40 PRINT"START":CALLA:PRINT"STOP"
 100 DATA38,A9,FF,85,70,85,71,85,72,C6,7
2,D0,FC,C6,71,D0,F6,C6,70,D0,F0,60
```

Fig. 6.12. A counting program that makes use of three bytes, all stored in zero page memory. (a) The assembly language program and (b) the BASIC poke program.

these addresses will cause havoc to the operation of the machine. A quick inspection of the memory shows that addresses \$7Ø to \$73 are often free, so perhaps we could make use of these for a really large count. Figure 6.12 shows the assembly language for this action. The three addresses which I have used are \$7Ø, \$71, and \$72. The program starts by loading \$FF into the accumulator, and then storing this number in all three zero page addresses. Because these are zero page addresses, we can make use of the fast and simple system of zero page addressing. The program makes no other use of the accumulator, so that the number \$FF will remain in the accumulator throughout. This allows us to store the accumulator in each memory address without reloading the accumulator.

The technique for the count-down is very much the same as before, except that there are three loops. You might try drawing a flowchart for yourself to see how these are arranged. The effect is that we are counting down the number 16777215 (denary) to zero. As you might expect, this takes a lot longer than a two-register count-down – several minutes. If you use a stop-watch to measure the time between the START and STOP messages, and divide this by the number shown above, you'll get some idea of how long (on average) each loop takes. Don't try to count-down this number in a BASIC program – life's too short!

Subroutine selection

We have looked at two types of loops in machine code, but completely neglected a third. A *holding loop* is one which keeps repeating a group of actions until some condition is satisfied. Suppose that you want a machine code program that takes the ASCII code for a key that has been pressed, and prints on to the screen the character corresponding to that key. A flowchart for this action is shown in Fig. 6.13. The first terminator is 'START', because every program or piece of program has to start somewhere. The arrowed line shows that this leads to the first 'action' block, which is labelled 'get character in A'. This describes what we want to do – get the code number for a character in the accumulator. The character may be zero, because no key has been pressed. This is where the holding loop comes in. The accumulator is tested, and if the byte in the accumulator is zero, the program loops back to the start. We don't know how we're going to do this at present – that comes later. After getting the character, the arrow points to the next

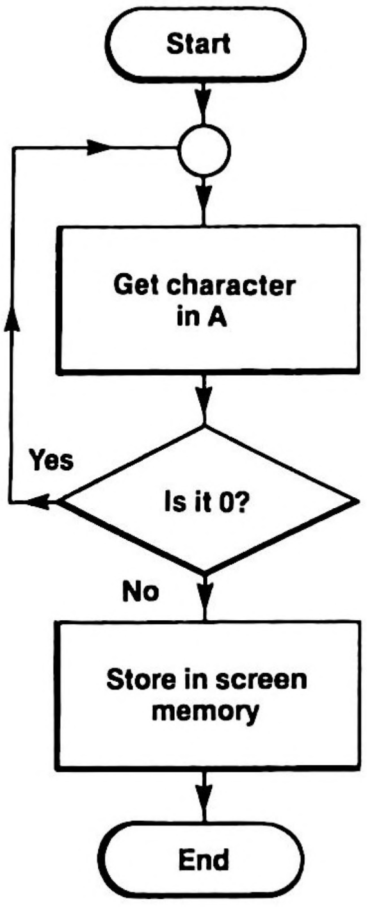


Fig. 6.13. The flowchart for a 'holding loop' action.

action, storing the byte in screen memory. That's how we carry out the 'print' part of the action, and it's something that we've looked at earlier. The END terminator then reminds us that this is the end of this piece of program; it's not an endless loop.

This is a very simple flowchart, but it neatly illustrates a holding loop action. Getting a character into the accumulator implies that the keyboard must be activated, looking for a key being pressed. If a key is not pressed, the keyboard must be checked again, over and over again. The next thing that we need to do is to find how we get a character into the accumulator from pressing a key. Routines of this sort already exist in the RAM of the Oric/Atmos, and we can use them. The same applies to placing a character on the screen. Converting each ASCII code number from the keyboard into the eight bytes that would produce the correct shape on the screen would be a considerable task. Once again, however, the routines already exist, and it would be pointless to write them all over again unless we wanted something really different.

Now a lot of computer manufacturers regard their ROM subroutines as a matter of secrecy. They discourage you from learning about them, and do not publish details of the subroutine addresses and how they should be used. Oric take the opposite attitude. The Oric/Atmos manual contains some useful details about the most useful ROM subroutines and how they should be used correctly. This way, Oric hope to encourage you to write machine code programs that will work well, and which will still work even if changes are made in the ROM at some later date. In addition, some subroutines are written so that you can 'break into' them – but more of that later.

The use of JSR

JSR is one of the 6502 mnemonics, which means 'jump to subroutine'. When the microprocessor comes across a JSR code, it will read the following two bytes as an address, place this address in the program counter, and start to execute the code at this new address. That's not quite all, though. Before the new address is placed in the PC, the old address (the address of the next instruction following the JSR) is stored temporarily. The idea is that JSR will cause a piece of code to be run, but at the end of that piece of code, an RTS instruction byte will cause the 6502 to return to the old address, carrying on as if the JSR had not been there. Sounds familiar? Of course it does, because it's the same as the GOSUB action in BASIC. JSR and RTS are to 6502 machine code as GOSUB and RETURN are to BASIC. The point about all the ROM routines that Oric tell you about is that each one ends with an RTS. If you call any of them into action with a JSR, then, you will find that your program goes on normally after the subroutine has been used. What you need to know, however, is what each subroutine does and what effect it has on the registers of the 6502. We'll start this quest by looking at two of the most useful subroutines, GTORKB and VDU.

No, these aren't misprints, just abbreviations. GTORKB means 'get a character from the keyboard'. The effect of the routine, then, is to read a character from the keyboard, rather like the action of KEY$ in BASIC. It carries out the flowchart action of the first part of Fig. 6.13, and needs to be followed by a holding loop. Figure 6.14 shows the assembly language version of the flowchart, with the address $EB78 used for GTORKB, and address $F77C for VDU. There are two points that we have to watch carefully, though. The first one is that the Oric-1 uses different addresses, $C5F8 for GTORKB and $CB61 for VDU. Secondly, the keyboard routine gets the characters as an ASCII code in the accumulator, but the

(a)

	Assembly language	Code
	ORG $96ØØ	———
LOOP:	JSR $EB78	2Ø 78 EB
	BEQ LOOP	FØ FB
	TAX	AA
	JSR $F77C	2Ø 7C F7
	RTS	6Ø

(b)
```
10 HIMEM#95FF:A=#9600
20 FORN=0TO9:READ D$
30 POKEA+N,VAL("#"+D$):NEXT
40 CLS:CALLA
100 DATA20,78,EB,F0,FB,AA,20,7C,F7,60
```

Fig. 6.14. The 'print a character' program (a) in assembly language and (b) as a set of POKEs.

screen printing routine needs the character as an ASCII code in the *X register*! How do I know that? The Atmos manual illustrates the use of these routines in Appendix 9. This is the sort of information that you *must* have if you are to use machine code. In the program, then, we must copy the byte in the accumulator into the X register. This is done by the TAX command (transfer A to X). The program is then delightfully simple – a call to GTORKB, the test for zero and loopback, a TAX, then a call to VDU and that's all! When this runs, pressing a key will cause a letter or other character to appear at the cursor position. This doesn't look any different from what happens when you press a key at any other time, so how do we know that this program works? Easy – just run it, and press a letter key. The letter will appear on the screen, with the READY prompt underneath it. Now press the RETURN key. The cursor simply moves down to the next line. There's no 'SYNTAX ERROR' message, as you would get if you simply typed a letter and then pressed RETURN. The 'SYNTAX ERROR' is missing because we have short-circuited the routine. We've broken a lot of new ground in this short piece of program, so perhaps this is a good time to go over it all carefully and make sure that you know what it has all been about before we plunge rather deeper into the business.

Testing your character

One thing that can be a bother here, as I have indicated, is that the Oric-1 may not use the same address. Several of the ROM addresses in the Atmos have changed as compared to the older Oric-1. The way round this is to use address $Ø23B for both the Oric-1 and the Atmos in place of $EB78. This address, $Ø23B, is only a sort of 'staging post'. It directs the subroutine to another part of memory, where there is more room. The point of using it is that $Ø23B is in RAM, and its contents can be altered. It's likely that if your machine does not keep its keyboard routine at $EB78, then the correct address will be kept at $Ø23B. If you use JSR $Ø23B in place of $EB78 then your program should still work. The screen print routine is similarly treated, with its address obtained by JSR $Ø238. Addresses $Ø238 to $Ø249 are used in this way. If you are using the Atmos, you can use the main ROM addresses that I have detailed here.

Meantime, back to the read and write routines. We can expand the idea that the program of Fig. 6.14 introduced. This time, we'll make the process repeat until the RETURN key is pressed. When the RETURN key is pressed, the ASCII code $ØD (denary 13) is placed in the accumulator by the GTORKB routine. What we have to do is test for this character, and jump out of the loop when it appears. Figure 6.15 shows the flowchart for this suggested program.

The complete program is illustrated in Fig. 6.16. The character which GTORKB has placed in the accumulator is compared with $ØD. The CMP step, remember, does *not* alter the size of the byte in the accumulator, but it will cause flags to be changed in the P (status) register. If the RETURN key has been pressed, the accumulator will contain the byte $ØD, and the CMP action will set the zero flag. If the zero flag is set, then the effect of BEQ OUT will be to make the program jump to the label word 'OUT'. If the RETURN key was *not*

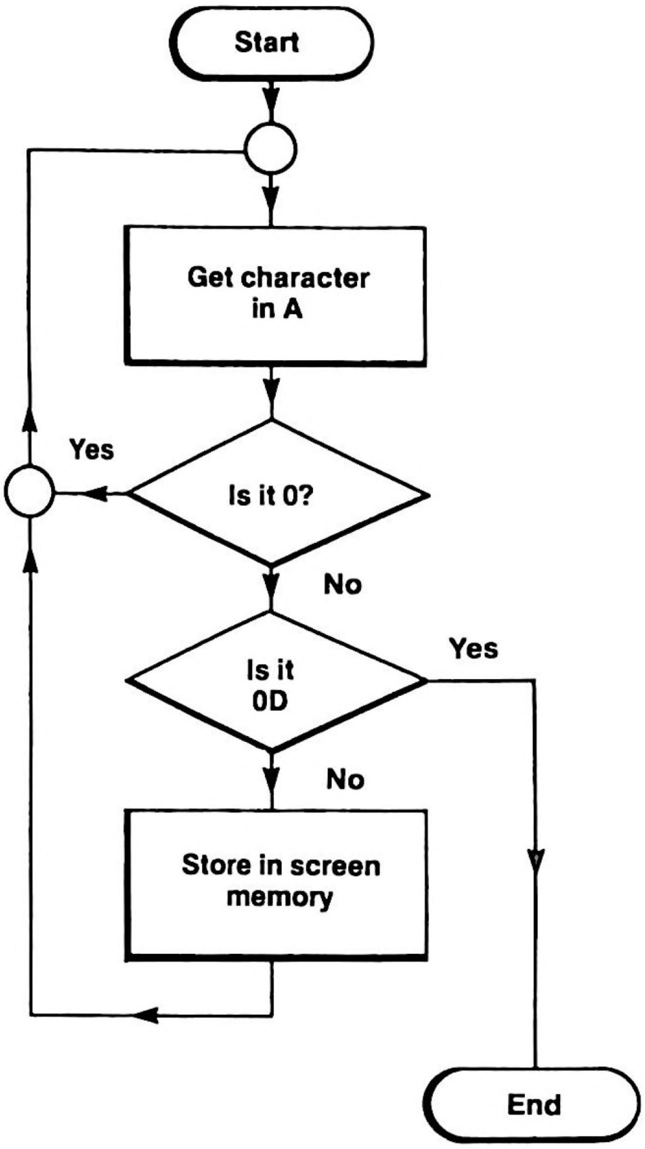


Fig. 6.15. A flowchart for a repeated character printing program. In this case, characters are printed until the RETURN key is pressed.

(a)

	Assembly language	Code
	ORG $96ØØ	———
LOOP:	JSR $EB78	2Ø 78 EB
	BEQ LOOP	FØ FB
	CMP #$ØD	C9 ØD
	BEQ OUT	FØ Ø7
	TAX	AA
	JSR $F77C	2Ø 7C F7
	JMP LOOP	4C ØØ 96
OUT:	RTS	6Ø

(b)
```
10 HIMEM#95FF:A=#9600
20 FORN=0TO16:READ D$
30 POKEA+N,VAL("#"+D$):NEXT
40 CLS:CALLA
100 DATA20,78,EB,F0,FB,C9,0D,F0,7,AA,20
,7C,F7,4C,00,96,60
```

Fig. 6.16. (a) The assembly language and (b) the POKE program for the character printing process.

pressed, then the jump is ignored; the JSR $F77C prints the character, then the JMP $96ØØ returns to the keyboard scanning routine.

From then on it's plain sailing. When the routine assembles, you can press keys to your heart's content, and see the result on the screen. You can then return to BASIC by pressing the RETURN key. What keys work? All the letter and number keys, including the DELETE, will have their normal effect. The arrowed keys will also allow the cursor to shift in the usual way. The CTRL key used with a letter key can have some odd consequences, however. Some CTRL sequences can cause the program to lock up, and you'll have to press the button under the computer to recover your program.

Meantime, what about trying some variations. What happens, for example, if the RETURN key is tested for *after* its code has been 'printed' by VDU? You will have to shift the CMP #$ØD and BEQ OUT steps so that they follow JSR F77C. Try it – can you explain what happens? It's just as if you had typed some letters at random, and then pressed RETURN. Not enough? Then try this – add between GTORKB and VDU the steps CLC and ADC #1. You will have to be careful about where you place CMP #ØD now. If CMP #ØD comes before the new steps, you can leave it alone. If it follows, then it will have to be changed to CMP #ØE. Try it, and you'll see why. Ever wanted to get into the message coding business?

Chapter Seven
Ins and Outs and Roundabouts

Video loops

Of all the loop programs that we can make use of in machine code, loops that involve the video display addresses are among the most useful. We have seen earlier that we can make things happen on the video display by making use of POKE commands from BASIC. The techniques that you used for POKE displays on the screen can also be used for machine code, and the main differences are that machine code is faster and involves more work on your part!

Take a look, for starters, at Fig. 7.1. This shows the flowchart for a program that will fill part of the text screen with one character. The idea is that we load a register (the accumulator is usually the best bet) with a code number for a character, and we store this at the first screen address, which is $BBA8. We must then increment this address, store the accumulator again, and repeat this process for as long as we need to, until we reach the last address. The flowchart shows what we have to do, but you need to know how to carry it out with the 6502. This is the kind of program which can make use of indexed addressing, so we'll start by recalling what this involves.

There are two index registers, X and Y, each of which can store a single byte number. When we carry out an indexed load or store (or any other operation that copies a byte from memory), the number that is held in the index register is added to the address that we have specified. This forms a new address, and it's this address that is used for the load or store. If, for example, we have 2 stored in the X register, and we specify a store as:

```
STA $BBA8,X
```

(hex numbers), then this means that the address which will be used is $BBA8+2=$BBAA, and the byte in the accumulator will be stored to address $BBAA. One of the features that makes this so useful is

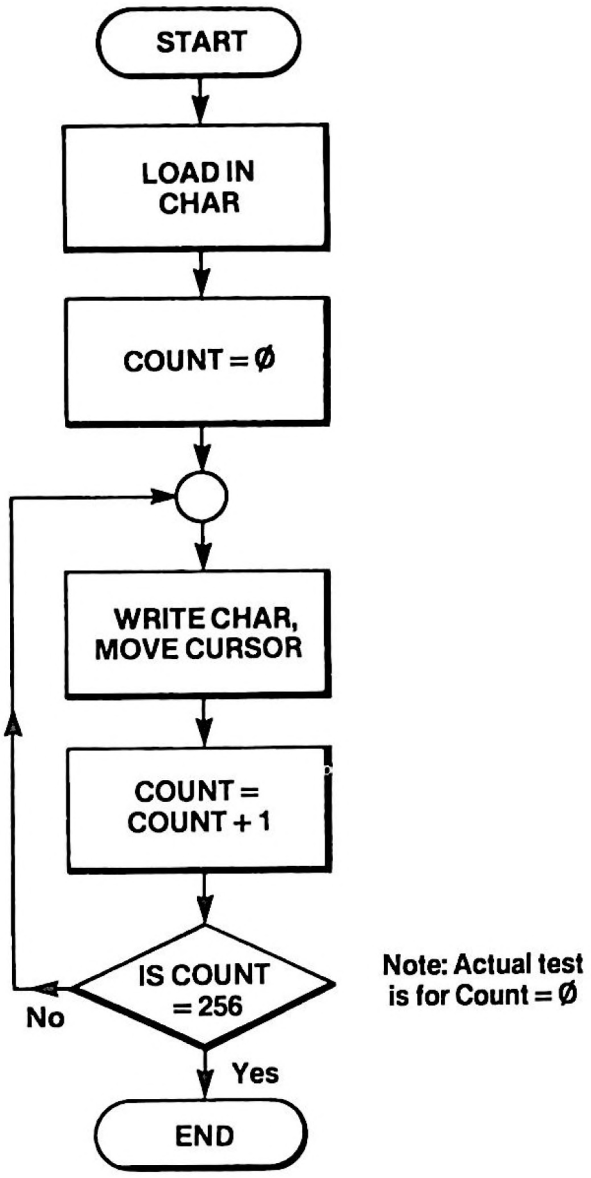


Fig. 7.1. A flowchart for filling part of the screen with a character.

that we have INX and DEX commands which will, respectively, increment or decrement the number that is held in the X index register. There are corresponding INY and DEY commands for the Y index register.

Let's get down to the assembly language version, which is shown, along with the BASIC program, in Fig. 7.2. The first step is straightforward – load the accumulator with $26. This is the ASCII code number that will produce the 'ampersand' sign, &. Obviously, you could try any other code number that you liked to use. We then load the X register with zero. The next item is the loop. We start the loop by storing the byte in the accumulator to the address given by $BBA8 (hex) plus the byte in the X register, and then incrementing

(a)

```
           LDA #$26           ;ASCII '&'
           LDX #$ØØ
LOOP:      STA $BBA8,X        ;put on screen
           INX                ;next address
           BNE LOOP           ;do next one
           RTS                ;back to BASIC
```

(b)

```
10 HIMEM#95FF:A=#9600
20 FORN=0TO10:READ D$
30 POKEA+N,VAL("#"+D$):NEXT
40 CLS:CALLA
100 DATAA9,26,A2,0,9D,A8,BB,E8,D0,FA,60
```

Fig. 7.2. (a) The assembly language program. (N.B. The code has not been shown this time because the assembly language is more important.) (b) BASIC poke program. This version, however, fills only the top part of the screen.

the X register. In assembly language, these steps are written as:

```
STA $BBA8,X
INX
```

The next step is to compare the content of the X register with zero, using BNE. Since the X register started at zero, and has just been incremented to 1, the comparison will not give zero, and so BNE will cause a loop back to store the character at another address. When the number in the X register is equal to $FF (255 denary), then the next increment action will make the content of the X register equal to zero again. This is because the register can hold only one byte. At this point, the BNE test fails, and the program breaks out of the loop, and returns to BASIC.

Now this does some of what we want, but not quite all. It will place a character in 256 screen addresses, but the text screen consists of 1080 addresses. What has limited us in this case is the size of the X register – one byte. This makes certain that an indexed load or store, carried out in a loop, cannot deal with more than 256 bytes. Working on the basis that half a loaf (or quarter of a screen) is better than nothing, we'll try it out. Later on, we'll see how we can get round this limitation.

Transforming this assembly language by hand into machine code is reasonably straightforward. Once the machine code bytes have been written down (check the displacement byte that follows the

BNE), then the BASIC program that pokes the bytes into memory can be written down (Fig. 7.2(b)). It shouldn't take long – apart from the value of N and the DATA line, it's almost the same as any of the other programs so far. At the end of the run, we must use the STOP and RESTORE keys to return the screen conditions to normal. When this runs, there is a short delay while the slow BASIC pokes the numbers into the memory, and then the machine code does its stuff in its usual lightning way.

The rest of the way

Writing a program that will fill all of the screen is not quite so easy, because of the limitation of the size of the index register. There are several ways of carrying out the action that we need, but the most straightforward method makes use of what is called *indirect addressing*. This was briefly mentioned in Chapter 4, and now that we're up against it, we'll have to take a closer look at one of the two methods that the 6502 can use. The two indirect addressing methods of the 6502 are often known as 'indexed indirect' and 'indirect indexed'. Titles like these are rather confusing, so in this book, I'll stick to the simpler descriptions of 'X-indirect' and 'Y-indirect'. This is because one method makes use of the X-index register, and the other method makes use of the Y-index register. The one we need for the loop program is the Y-indirect method.

The way that Y-indirect addressing operates is as follows. The first address that we want to use, such as $BBA8 in our example, is stored as two bytes in two consecutive zero page addresses in memory. As always, the bytes are stored in the low-then-high order, and they *must* be in zero page memory. A number is also placed in the Y index register. Now when we carry out a memory operation such as a load or a store, we use the Y-indirect form. In assembly language, a Y-indirect store would be written in the form:

```
    STA (address),Y
```

Its effect is fairly complicated at first sight. The low-byte of the address is copied from the memory, and the content of the Y-register is added to it. If there is any carry from this addition, it is added to the high-byte. The two new bytes of address are then used for the store (in this example) operation. Suppose, for example, that we stored the first screen address of $BBA8 in zero page memory, using hex addresses $25 and $26. Using the low-high order of storing means that address $25 will store $A8 and $26 will store $BB. Now if we have the number $2Ø (hex) stored in the Y register, then the effect of:

STA ($25),Y

will be to add $2Ø to the number stored in $25 (which was A8), and so form the number $C8. The upper part of the address then comes from $26, giving the complete address as $BBC8. This is the address to which the byte in the accumulator will be copied.

(a)

```
          LDY #$A8        ;add to screen address
          LDA #$ØØ        ;screen start low byte
          STA $25         ;store at $25
          LDA #$BB        ;screen start high byte
          STA $26         ;store at $26
          LDA #$2A        ;load character &
LOOP:     STA ($25),Y     ;store at screen address
          INY             ;increment address
          CPY #$DF        ;is it low byte of end?
          BEQ TEST        ;check high byte if so
          CPY #$ØØ        ;check need to increment high byte
          BEQ BUMP        ;do it if needed
          BNE LOOP        ;otherwise back and repeat
TEST:     LDX $26         ;get high byte
          CPX #$BF        ;compare high byte, last address
          BNE LOOP        ;back if not
          RTS             ;out if so
BUMP:     INC $26         ;increment high byte
          BNE LOOP        ;back to loop
          RTS             ;out if anything goes wrong
```

(b)

```
10 HIMEM#95FF:A=#9600
20 FORN=0TO36:READ D$
30 POKEA+N,VAL("#"+D$):NEXT
40 CLS:CALLA
100 DATAA0,A8,A9,0,85,25,A9,BB,85,26,
A9,2A,91,25,C8,C0,DF,F0,6
110 DATAC0,0,F0,9,D0,F3,A6,26,E0,BF,D
0,ED,60,E6,26,D0,E8,60
```

Fig. 7.3. Filling the entire screen. (a) Assembly language. (The hex codes are not shown now, so that you can concentrate on the assembly language.) (b) BASIC poke version.

Figure 7.3(a) shows the complete assembly language version of the program. The first six steps place the correct values into the registers and into memory. This has not been done in the most efficient way, but at least it's easy to follow. Where things start to get more difficult is step 7, at the start of the loop. What causes the problem is the need to carry out the incrementing of the screen address for 1079 times. We'll take a look at this part of the program in close detail.

At the start of the loop, the number in the Y register is $A8, and the 'base address' for the STA operation is stored in zero page addresses $25 and $26 (hex). The address $25 stores ØØ and address $26 stores $BB, so that the two bytes, plus the byte in the Y register, make up the first screen address of $BBA8, denary 48Ø4Ø. When the STA ($25),Y step is used for the first time, then, the byte in the accumulator will be put into address $BBA8. The INY step then increments the number in the Y register from $A8 to $A9. Two comparison steps then follow. The first one compares the contents of the Y register with $DF. This is because the last screen address is $BFDF. We don't want to stop the loop just because the Y register holds $DF, though, only when the address $26 happens to be storing $BF at the same time. The BEQ TEST stage, then, jumps to a piece of program that will test the number in address $26 to find if we have reached the end of the screen. If we haven't, then the BNE LOOP at the end of the TEST routine will get back to the loop.

There's another test, however. Because an address can hold only one byte, we have to look for the Y register counting past $FF to $Ø. When this happens, we must increment the upper part of the address, so that the address which follows $BBFF is $BCØØ rather than $BBØØ. This is done by the second check, CPY $#ØØ. If the Y register contains zero, then the program goes off to the BUMP routine. This simply increments address $26, and then goes back to the loop. The return to the loop could have been done by using JMP, with a full address, but it's more convenient to use BNE. The content of $26 should never get to zero in the course of the program, so the jump is really unconditional. Just in case of mishaps, though, I have followed it with the RTS. That's called defensive programming. Fig. 7.3(b) shows the BASIC poke program which places the bytes in memory, and runs the machine code.

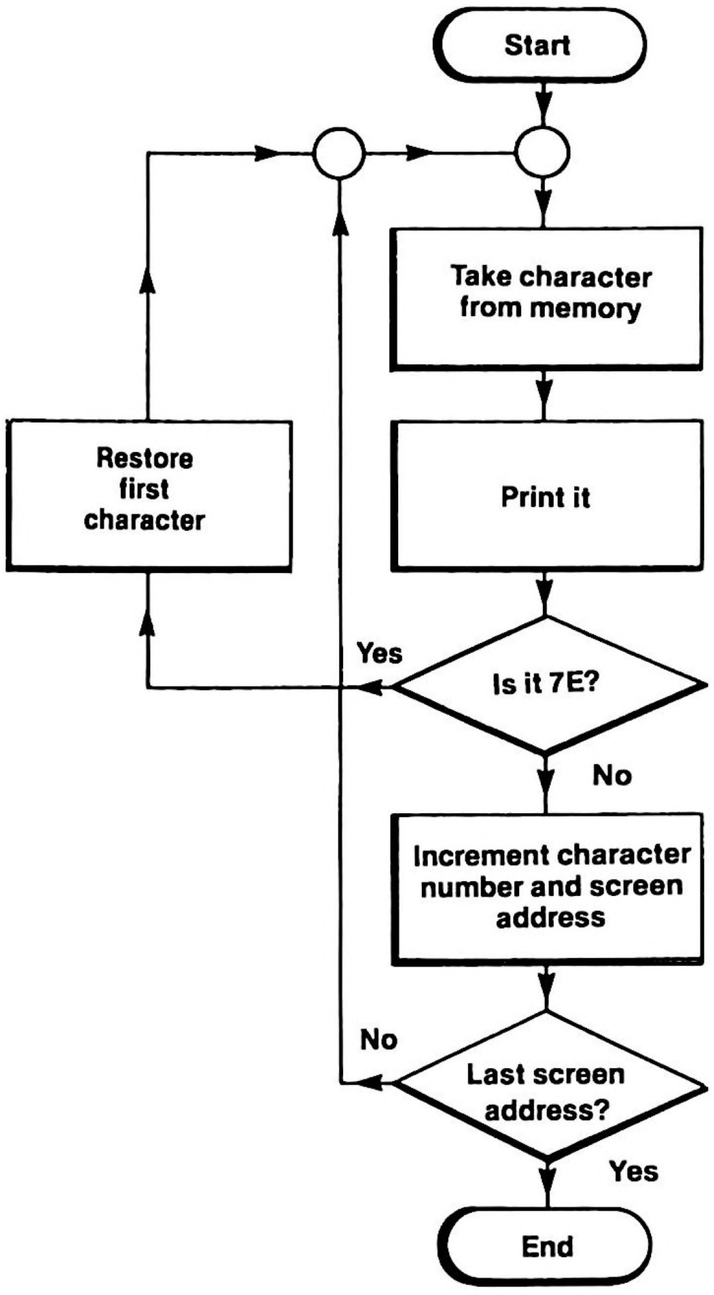


Fig. 7.4. A flowchart for printing the entire character set, ASCII codes $2∅ to $7E.

Take a bigger cast-list

We can modify the program of Fig. 7.3 in an interesting way. Suppose we started with the accumulator containing $2∅, and we incremented the accumulator on each pass through the loop. This would mean that we would produce on the screen each character for the numbers $2∅ to $FF (255 denary). What would happen then? Well, since the accumulator can hold only eight bits, and $FF is the largest number it can hold, it simply goes back to ∅ next time it is incremented. We don't really want to place some of these numbers in the screen memory, however. The numbers below $2∅ will cause colour (and other) effects, the numbers above $7E (127) will give some similar effects. To obtain a display of all the normal characters of the Oric/Atmos, then, we should restrict the numbers in the accumulator to the range $2∅ to $7E. Figure 7.4 shows the flowchart for this action, and Fig. 7.5 shows the assembly language and the BASIC poke program. There's quite a lot in this program, and it's about as long a machine code program as I would care to write without the aid of an assembler. Let's look at it in detail.

The first seven steps are concerned with loading registers and

```
          LDY #$A8          ;lsb start
          LDA #$∅∅          ;lsb base address
          STA $∅5           ;put into page ∅
          LDA #$BB          ;msb start
          STA $∅6           ;page ∅
          LDA #$2∅          ;first character
          STA $∅7           ;put in page ∅
LOOP:     LDA $∅7           ;get character
          STA ($∅5),Y       ;indirect store
          CMP #$7E          ;is it last character?
          BEQ RSTOR         ;restore $2∅ if so
          INC $∅7           ;pick next character
NXT:      INY               ;next address
          CPY #$DF          ;check for end
          BEQ TEST          ;with subroutine
CTN:      CPY #$∅∅          ;new high byte needed?
          BEQ BUMP          ;if so, do it
BACK:     BNE LOOP          ;repeat actions
TEST:     LDX $∅6           ;take high byte
          CPX #$BF          ;end of screen?
          BNE CTN           ;back if not
          RTS               ; out if it is
BUMP:     INC $∅6           ;new high byte
          BNE BACK          ;back to main loop
          RTS               ;just in case
RSTOR:    LDA #$2∅          ;start again
          STA $∅7           ;back in page ∅
          BNE NXT           ;back to main loop
          RTS               ;just in case
```

(Fig. 7.5(a))

(b)
```
10 HIMEM#95FF:A=#9600
20 FORN=0TO53:READ D$
30 POKEA+N,VAL("#"+D$):NEXT
40 CLS:CALLA
 100 DATAA0,A8,A9,00,85,5,A9,BB,85,6,A
9,20,85,7,A5,7,91,5
 110 DATAC9,7E,F0,19,E6,7,C8,C0,DF,F0,
6,C0,0,F0,9,D0,EB,A6,6
 120 DATAE0,BF,D0,F4,60,E6,6,D0,F3,60,
A9,20,85,7,D0,E3,60
```

Fig. 7.5. (a) The assembly language version, (b) the BASIC listing for the flowchart of Fig. 7.4.

memory. The starting address of the text screen is loaded into addresses $25 and $26, plus Y register, as before, but this time address $27 is also used to hold the first character code of $2∅. The loop starts by loading the accumulator from this address, and using Y-indirect addressing to store it in the screen memory. The next step tests what this number in the accumulator is. If it is less than $7E, well and good. If not, then the RSTOR routine will restore the value of $2∅ in address $27 and resume normal service at the point which is labelled NXT.

Having dealt with this problem, the rest of the program follows lines which should be more familiar to you. The address $27 is incremented, so that we get the next ASCII code number, and the Y register is also incremented. We now have to test for the last screen address, just as we did in the program of Fig. 7.3, and for the Y register reaching the value of ∅. These loops follow the same pattern as before. Try it, and you'll see the full set of text-mode characters appearing, repeated several times over the whole screen. This is a good example of how a simple program can be extended so as to do much more than the original version. It's an important point, because a lot of machine code programming is of this type. If you keep a note of all the assembly language programs that you have ever used, along with what they did, then you'll find that this 'library' is a very precious asset. Very often, you'll find that any new program that you want to write can be done by modifying or combining (or both) old routines that you are familiar with. Another big advantage of this is that an old routine is a trustworthy routine – a split-new one needs a lot of testing before you can rely on it.

Save it!

At this point, when our programs are getting noticeably longer, and doing more interesting things, it's time to look at the topic of saving machine code program on tape. Now you aren't in any way obliged to do this. Our machine code programs have all been, so far, in the form of a BASIC program that poked numbers into the memory. You can, obviously, just save this BASIC program, and it will create the machine code for you whenever you want. When you are using a mixture of BASIC and machine code, this is the ideal way of saving and loading the machine code bytes. There are times, however, when you need as much of the memory as possible, and another piece of BASIC program is as welcome as an elephant in a space capsule. You may also want to put on cassette a program that is a mixture of BASIC and machine code, so as to make it difficult for anyone to copy it. Either way, you'll want to make a direct recording of the bytes that are stored in the memory. The ordinary BASIC commands of the Oric/Atmos can cope with this, but, as we'll see, in a slightly different way.

Like most other machines, the Oric/Atmos has special BASIC commands for saving and reloading machine code programs. These use the familiar CSAVE and CLOAD commands, but with a different syntax. The idea is that the CSAVE command will save a list of bytes, all the bytes that are stored between a starting address and a finishing address. The CLOAD command, similarly, will load a list of bytes to the same range of addresses as they were saved from. You can also specify that your machine code program will run as soon as it has been loaded, which saves you from having to remember which address number to use with CALL.

Let's try it out. Use the program of Fig. 7.5 to create a machine code program. The starting address is $96ØØ, and the finishing address is $9635. This is because the program poked bytes from Ø to 53 (denary) and 53 denary is $35 hex. The CSAVE command must therefore read:

```
CSAVE"SCRCOVER",A#96ØØ,E#9635
```

and if you want the program to run as soon as it is loaded, you add another comma and the word AUTO. Start your recorder and press the RETURN key (as fast as possible after starting the tape), and the machine code will be recorded. This takes less time than recording the BASIC poke program.

Loading a program of this type is much easier. You don't have to specify any address numbers, particularly if you use AUTO. Simply type:

```
    CLOAD"SCRCOVER"
```

and press RETURN. Your machine code program will load and, if you used AUTO, will run also. If you didn't use AUTO, you will have to start the machine code with a CALL, using the correct starting address. A machine code load with auto-run is very difficult to disable, and it's possible to make a machine code program which will then load and run a BASIC program. This is a good way of protecting your programs against copying – but more of that later.

Don't imagine, by the way, that this ability to save and load bytes applies only to machine code programs. Any section of the memory can be saved and reloaded by using this method. Since the text screen is controlled by the bytes in memory from $BBA8 to $BFDF, you can CSAVE and CLOAD text and LORES screen patterns in this way. You can similarly load and save HIRES screen patterns by using addresses $A∅∅∅ to $BF#F. The patterns on the LORES screen will be rather spoiled by the READY message that appears after loading, though, so for best results, you should disable this by using a piece of program, such as a closed loop, that stops the end of a program.

Take a message

After that brief interlude on the subject of saving and reloading machine code, let's get back to the programs. We left Fig. 7.5, you remember, writing characters on the screen. It's time now that we looked at ways of putting something more interesting on the screen, and letters look like a reasonably simple start to this type of programming. What do we have to do? Well, to start with, we need to store some ASCII codes for letters somewhere in the memory; we can't just use a string variable as we would in BASIC. We will have to know the address at which the first of the letters is stored, and how many letters are stored starting at this address. After that, we should be able to work a loop which takes a byte from the 'text space' (where the letter codes are stored) and put it into the screen space (the screen addresses). We've already used the main type of instruction that we need for this sort of thing – the auto-incremented load or save. To work, then.

We start, as always, with a flowchart. It's not so easy this time, because we need a different way of ending the loop. We could count the number of letters that we want to place on the screen, but I want to look at a different technique this time – using a terminator. You are probably familiar with this idea used in BASIC programs. A 'terminator' is a byte which the program can recognise as a special character, one which is not, for example, part of a message. A convenient terminator for a lot of purposes is Ø, so we'll try that. The difficulty arises because we don't want this terminator printed on the screen. As it happens, because of the way that the Oric/Atmos uses its code numbers, the number Ø actually would not cause any bother, but we'll avoid printing it all the same. This means that we must test the accumulator between loading the byte from memory and placing it into the screen memory. That, as you'll see, makes the loops slightly more complicated.

The flowchart that we need is shown in Fig. 7.6. What we have to

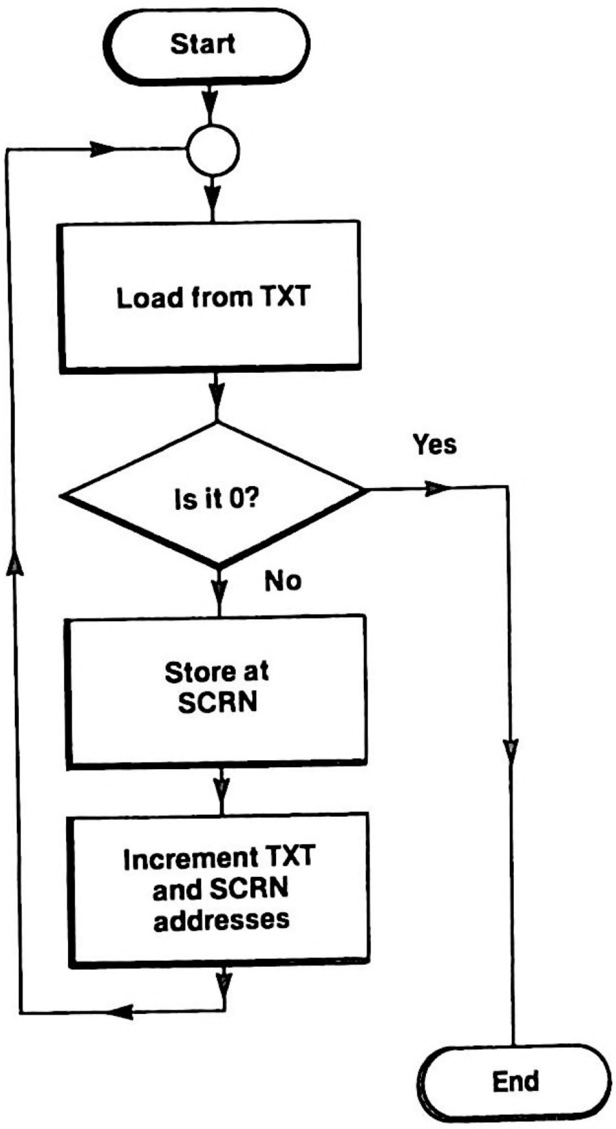


Fig. 7.6. A flowchart for printing a message on the screen.

do is to store two addresses. One of these will be familiar, it will be part of the screen memory. This has to be the address of the first byte that we will want to put on the screen. The other address has to be a store address, the start of a string of bytes that will be used to store ASCII codes, and which will not be used for anything else. We can clear things up for ourselves by giving these two addresses label names. I've chosen SCRN for the screen memory, and TXT for where we're storing the codes. What we do, then, having allocated these addresses, is to load a code from TXT, and increment the TXT address. We then test the character, to see if it's our terminator of ∅. If it is, we want to leave the program at once. If it's not, then we store this byte at SCRN, increment the SCRN address, and go back for another character. Now this gives us a flowchart which has two jumps. One of these is the 'go back' part, the other is the part that goes to the end. What is this going to look like in assembly language?

(a)

```
TXT EQU $965∅
SCRN EQU $BDBB

          LDX #$∅∅        ;zero X register
LOOP:     LDA TXT,X       ;get text character
          CMP #$∅∅        ;is it end?
          BEQ OUT         ;end if so
          STA SCRN,X      ;store to screen
          INX             ;pick next one
          BNE LOOP        ;go round again
OUT:      RTS             ;back to BASIC
```

(b)

```
10 HIMEM#95FF:A=#9600:B=#9650
20 M$="This is your life!"
30 FORN=0TO17:POKEB+N,ASC(MID$(M$,N+1
,1)):NEXT
40 POKEB+18,0
50 FORN=0TO15:READ D$
60 POKEA+N,VAL("#"+D$):NEXT
70 CLS:CALLA
100 DATAA2,0,BD,50,96,C9,0,F0,6,9D,BB
,BD,E8,D0,F3,60
```

Fig. 7.7. (a) The assembly language for the message routine, (b) the BASIC listing.

The answer appears in Fig. 7.7(a). It follows the flowchart pretty exactly, and the parts we particularly need to look at are the BEQ and the BNE steps. At the BEQ step, the accumulator has been loaded from the TXT piece of memory, whose starting address is $965ø. I have picked an address which is well clear of the addresses that we are using for the program. The accumulator is loaded using X-indexed addressing, and the X register has been loaded with zero. As a result, the first load of the accumulator will come from the address $965ø. This will be the byte $54, which is the ASCII code number for the letter 'T'. The CMP #ø step is put following the load so that the ø byte can be detected at the end of the message. BEQ means 'branch if equal to zero', and the displacement that follows this instruction byte will take the program to the RTS instruction, labelled OUT, skipping over the steps between the BEQ and the RTS. If the byte is not zero, however, it is stored at the SCRN address, using the X register once again for indexing. We then have to get back for another character. Since this needs a jump, and one that must always be made at this point, we could use the JMP instruction. JMP, however, has to be followed by a full two-byte address, and it's a lot easier to use BNE. At this point in the program, the byte in the X register can never be zero (unless you're trying to place too many letters on the screen). As a result, BNE will always return the program to the LOOP position. This will continue until a zero is loaded into the accumulator, and the BEQ OUT step forces the program to return to BASIC.

That explained, we can convert into the form of a BASIC program, as Fig. 7.7(b) shows, and try it out. We'll place the bytes into memory in a fairly simple way, by poking them into memory from a string. This is done in lines 2ø to 4ø. Lines 5ø and 6ø then put the machine code into place, and line 7ø runs it. When it runs, you see the message appearing.

Perhaps we can deal now with one other point about placing text on the screen. In all of the preceding programs that use text on the screen, we have simply accepted the black and white text that always appears when we use direct poking of numbers into screen memory. We can, of course, do a lot better. What we shall do is to place some colour codes into memory as well, so that the screen action will be affected by these as well. We can use both foreground and background colour codes, remembering that each colour code number will affect all the letters that follow it on a line.

This sort of thing needs only a comparatively small change to a machine code program. Figure 7.8 shows the BASIC version. Line

```
 10 HIMEM#95FF:A=#9600:B=#9650
 20 M$="This is your life!"
 25 POKEB,19:B=B+1:POKEB,4:B=B+1
 30 FORN=0TO17:POKEB+N,ASC(MID$(M$,N+1
,1)):NEXT
 40 POKEB+18,0
 50 FORN=0TO15:READ D$
 60 POKEA+N,VAL("#"+D$):NEXT
 70 CLS:CALLA
 100 DATAA2,0,BD,50,96,C9,0,F0,6,9D,BB
,BD,E8,D0,F3,60
```

Fig. 7.8. Modifying the routine so that the colour addresses are poked.

25 has been added to poke two colour code numbers into the memory ahead of the text message. This gives a yellow background and blue lettering for your text. In a machine code program which produced text on the screen, of course, you would not have a string used to place the characters. You would have to place the ASCII codes in memory before you recorded the machine code. This could, however, be done in exactly the way as is shown in Fig. 7.7 and Fig. 7.8. It's another small step forward!

Sailing out of the port

When I described the action of the computer system in Chapter 1, the idea of a *port* was raised. As far as a computer is concerned, a port is any chip or collection of chips that carries out the actions of sending bytes out or taking bytes in. As it happens, the port arrangements of the Oric/Atmos are documented in the manuals (particularly so in the Atmos manual), but you would not normally need to make use of them unless you wanted to arrange machine-control or other interfacing projects. This type of work is rather more complicated, and it's beyond the scope of this book. The reason is that the port chip is as complicated a device as the microprocessor itself, with its own registers and memory addresses, and it would need a book as long as this one to explain its actions fully, starting from scratch. Once you have had some experience of machine code programming, however, you will be able to take port programming in your stride, and you might find my book *Practical Microprocessor Systems* (published by Newnes) of use to you. It's even available in Japanese translation!

Chapter Eight

Debugging, Checking and Oricmon

Debugging delights

Now that you have experienced some of the delights of machine code programming, it seems fair to mention some of the drawbacks. One of these is debugging. A 'bug' is a fault in a program, and debugging is the process of finding it and eliminating it. It all sounds rather insecticidal, but it's nothing like as easy as that!

It's easy to say, I know, but the first part is prevention. Check your flowchart carefully to make sure that it really describes what you want to do. When you are satisfied with the flowchart, turn to the assembly language to make sure that it will carry out the instructions of the flowchart. When you are happy with this, check that the bytes you intend to poke into memory are the bytes which correspond to the assembly language instructions. One thing to watch very carefully is that you have the correct code for the addressing method that you are using. If you check each stage in the development of a program in this way, you will eliminate a lot of bugs before they are up and flying. Don't feel that you are a failure if the program still doesn't run – unless a machine code program is very simple, there's a very good chance that there will be a bug in it somewhere. It happens to all of us – and it's only by experience that you can get to the stage where the bugs will be few in number and easy to find.

If you use a good assembler, one source of bugs completely disappears. Human frailty means that the process of converting assembly language instructions into machine code bytes is error-prone. That's because it means looking up tables, and anything which involves looking from one piece of paper to another is highly likely to introduce mistakes. I shall briefly describe the action of the Oricmon assembler later in this chapter. At the time of writing, there were several assemblers available for the Oric/Atmos, but Oricmon has several advantages, two of which are that it is itself written in machine code and that it contains several valuable monitor actions. There's more on monitors later in this chapter as well! If machine code has really caught your imagination, and you feel that you want to branch out into more advanced work than we have space for in this book, then a good assembler and monitor program is an essential. You will have to be prepared to pay quite a lot for such a program. If, however, you intend to be just a dabbler, spawning the odd drop of machine code now and again, then the poke-to-memory methods that we have used so far will be perfectly adequate. Using these methods, however, means that there will be bugs lurking in each corner of the code. The main cause of these bugs is weariness. Converting an assembly language progam into hex bytes, and writing them in the form of DATA lines for a BASIC poke program is a tedious job, and all tedious jobs result in mistakes (ever driven a 'Friday' car?). Faulty address methods are one common result of tedium, and simply writing down the wrong code is another. One very potent source of trouble is with branch displacements. You may get the number wrong somewhere between subtracting addresses and converting a number (particularly a negative number) to hex. Another problem arises when you modify a program, and add code between a jump instruction and its destination. Having done that, you then forget to alter the size of the displacement byte! This is a problem which simply doesn't arise when an assembler is used. An incorrect jump will nearly always cause the computer to lock up. You can often restore control with the use of the RESET button, but not always, and you will sometimes lose your program (you did record it, didn't you?). Another form of incorrect branch is doing the opposite of what you intended, like using BEQ in place of BNE or the other way round. Careful thought about what the jump will do for different sizes of bytes should eliminate this one.

A lot of problems, as I have already said, can be eliminated by meticulous checking, and it pays to be extra careful about branch displacements, and about the initial contents of registers. A very common fault is to make use of registers as if they contained zero at the start of the program. You can never be certain of this. It's safer, in fact, to assume that each register will contain a value that will drive the computer bananas if it is used. With all that said, and with all the effort and goodwill in the world, though, what do you do if the program still won't run?

There's no single, simple, answer. It may be that your flowchart doesn't do what you expect it to do, and if you didn't draw a flowchart, then you have got what you deserve. It may be that you are trying to make use of an Oric/Atmost ROM routine and it doesn't operate in the way that you expect. When you have enough experience in machine code to follow more elaborate programs, you can make use of a disassembly of the ROM. At the time of writing, there is no disassembly of the Atmos ROM. There was one available for the older Oric-1, in a book called *The Oric-1 Companion*, by Bob Maunder (published by Linsac). Once again, this is for the really serious Oric/Atmos machine code programmer. All I can do here is to give you general guidance on removing the bugs from a program that seems to be well-constructed but which simply doesn't work according to plan.

The first golden rule is never to try out anything new in the middle of a large program. Ideally your machine code program will be made up from subroutines on tape, each of which you have thoroughly tested before you assembled them into a long program. In real life, this is not so easy, particularly when the subroutines exist only as DATA lines for BASIC poke programs. As usual, users of an assembler have the best of it, because they can keep assembly language instructions stored like BASIC programs, and merge and edit them as they choose.

The next best thing to keeping a subroutine library on tape is to have extensive notes about subroutines. In addition to routines of your own, you can keep notes on routines which you have seen in magazines. *Personal Computer World* runs a series called *SUBSET*. This consists of several general-purpose machine code routines each month. Most of these are for the two most-used microprocessors, the Z80 and the 6502. Even if you don't use the routines, the way in which they are documented should give you some ideas about how you should keep a record of your own routines - I personally would buy the magazine for this feature alone! If you are going to use a new routine in a program, it makes sense to try it on its own first so that you can be sure of what has to be placed in each register before the routine is called, and what will be in the registers after. Look at the examples in *SUBSET*, and see how well this information is presented. (The editors of *SUBSET*, Alan Tootill and David Barrow, have recently published *6502 Machine Code for Humans* (Granada) which is recommended as further reading after you have finished this book.)

Planning of this type should eliminate a lot of bugs, but if you are still faced with a program that doesn't work, and which you don't want to have to pull apart, then you will have to use breakpoints. A breakpoint, as far as the Oric/Atmos operating system is concerned, is the byte $6∅. This is the RTS byte, and its effect is to return to BASIC. When you are back in BASIC, you can examine the contents of memory by using PEEK instructions. The principle is to pick a point in the program at which something is put into memory. If you place a $6∅ byte following this, then when the program runs, it will return to BASIC immediately after the memory is used. By using a PEEK, you can then check that what has been loaded into the memory is what you expect. If it isn't, you should know where to look for the fault. If all is well at this point, then substitute the original byte that belongs in place of the $6∅, and place the $6∅ at the next address following a memory store command. This type of action, however, is much more easily carried out with the help of a good monitor program, of which more later.

The most awkward fault to find by this or any other method is a faulty loop. A faulty loop always causes the computer to lock up. When this happens, pressing the RESET button under the case of the Oric/Atmos will not normally get you back to normal service. You will then have to switch off and then on again, losing the program. You'll be glad then that you recorded the program before running it! The main cause of this sort of thing is a loop back to the wrong position. For example, if we had a program part of which read:

```
         LDX,#$FF
LOOP:    DEX
         BNE LOOP
```

we could encounter problems. Suppose that this was assembled by hand, and we made the branch back to the LDX instruction rather than to the DEX instruction. This would result in the X register being kept 'topped up', and never decremented to zero, so that the loop would be endless. A mistake like this is easily spotted in assembly language, because the position of the label name is easy to check. It is very much more difficult to find when you have only the machine code bytes to look at. As always, taking care over loops is the only answer, and the method that has been shown in this book of calculating and checking displacements is a good precaution.

Monitors

I mentioned monitors briefly earlier on in this chapter. This has nothing to do with a TV monitor, which is a sort of superior quality

TV display for signals. A monitor in the software sense is a program, one which checks (or monitors) each action of a machine code program. A monitor is (or should be!) a machine code program which can be put into the memory at a set of addresses that you aren't likely to use for anything else. Once there, a monitor allows you to display the contents of any section of memory (in hex), alter the contents of any part of RAM, and inspect or alter the register contents of the 6502. These are the most elementary monitor actions, and it's useful if a section of program can be run, breakpoints inserted, and registers inspected on a working program. The ideal monitor would be one which could carry out the steps of a machine code program one at a time, displaying the register and memory contents at each step. Oricmon comes very close to this ideal.

An even better choice is a combined monitor, assembler and disassembler. The monitor allows you to debug programs, the disassembler lets you find out how ROM routines (and other machine code programs) work, and the assembler allows you to assemble correct code into memory from your assembly language programs. Unfortunately, you have to be careful about your choice of such programs. Some are in BASIC, which means that they are slow, very long, and they occupy a piece of memory that you may need. One that I sent for ('48-hour delivery guaranteed', it said) had not arrived in three weeks, and when it did, it refused to load into the Atmos. Another arrived fast enough, but was in BASIC, contained bugs (like numbers which were printed in background colour!) and took forever to load. It also used very non-standard mnemonics, which meant that every assembly language program had to be altered. Tansoft produce a combined monitor and disassembler which has some assembly actions. This program is called Oricmon, and the remainder of this chapter is devoted to the facilities which are available in this program. I quote from the Oricmon manual here because my copy of Oricmon would not run in my Atmos, though it appeared to load. It was a tape that was intended for the Oric-1 and had not been modified.

Oricmon monitor

The Oricmon assembler/monitor contains among its many facilities an excellent monitor. The program should be self-starting, and its monitor facilities should be available to you whenever it has loaded.

These facilities are obtained by a set of simple commands, which you can very soon memorise, so that it will not take very long to become accustomed to the use of Oricmon. The facilities are, in order of presentation in the manual, memory inspection, change data, enter text, hex arithmetic, move data, compare memory, and return to BASIC. The use of the 'P' key allows you to have results printed on paper (if you have a printer) or not as you choose. In addition to the monitor facilities, Oricmon allows disassembly (translating code into assembly language) and a simple assembly system. An important point to note is that *all numbers are assumed to be in hex*, so you will need to brush up your hex in order to use Oricmon.

Memory magic

One of the most-used actions of a monitor is to check what has been stored in memory. You can, of course, do this in BASIC by making use of PEEK, but the monitor action is much more flexible and useful. The main point is that you need only a brief command when you use the monitor, and you can inspect as much memory as you want with very little effort. If you want to see what is stored at only one location in memory, you simply type the address *in hex*, and press the RETURN key. The address will then be printed at the left-hand side of the screen, with a colon following it, and the byte that is stored at this address following the colon. You can then inspect the memory from this point on simply by pressing the RETURN key. Each time that you press the RETURN key, you will see displayed on the screen a starting address, and the next eight bytes that are stored in the memory. For example, the display of bytes from #9ØØØ might look like:

```
    9ØØØ: A9 ØC 2Ø 85 93 A2 ØØ BD
```

which shows the bytes that are stored in memory locations $9ØØØ to $9ØØ7. The next line (press RETURN again) would show the bytes for addresses $9ØØ8 to $9ØØF and so on.

Very often when you are inspecting memory in this way, you may be looking for ASCII codes. To make this simpler, Oricmon allows you to vary the display of memory. When you first use the display, you will see hex codes displayed. By pressing the Z key, you can change the display to ASCII, so that each ASCII code will print as its ASCII *character* instead of as a hex code. In this mode, any number that is not an ASCII code will be displayed as a dot.

Pressing the Z key again will give mixed display, with all ASCII codes shown as characters, and all others as hex codes. This is useful if you have a piece of memory which contains a mixture of commands and text. Pressing the Z key again restores hex printouts.

If you want to display a block of memory between two addresses, then you can obtain this in a simpler way than by pressing the RETURN key. You simply type the start address, a dot, and the finish address, then press RETURN. For example, typing 9ØØØ.9Ø8Ø and then RETURN will display the contents of memory between these addresses. There must be no spaces between each address and the dot. If, having obtained this display, you then find that you want to see some more, you can add another ending address by typing it following a dot. For example, typing .9ØAØ (RETURN) following the previous example will make the display continue to this new address.

Entering data

Another useful monitor facility is that of altering data bytes in the memory. Again, this is something that you can do from BASIC by using POKE, but the monitor allows more convenient control over what you are doing. Suppose that you want to enter the bytes AØ C9 at address $96Ø2. You would type:

```
    96Ø2:AØ C9
```

and then press RETURN. There must be no space either side of the colon, but you *must* put a space between the two data bytes. You can, of course, enter more than two data bytes in this way, but you must be sure to type a space between each byte and its neighbours. Each data byte should consist of two characters.

Text can also be entered in this way, which is very convenient if you are writing a machine code program which displays text on the screen. Text is entered by typing the address at which you want the text to start, then a single apostrophe ('), the text, and another apostrophe. Note that no colon is used in this case. There must, as usual, be no space between the end of the address and the first apostrophe. After you have typed the second apostrophe, press RETURN. If, for example, you wanted to have the ASCII codes for 'THIS IS YOUR LIFE' stored starting at address $96ØØ, you would type:

```
    96ØØ 'THIS IS YOUR LIFE'
```

and then press RETURN.

Hex arithmetic

Unless you possess the machine code programmer's ultimate status symbol, a calculator which works in hex, it's likely that you will avoid hex arithmetic like the plague. Oricmon allows you to carry out hex arithmetic reasonably painlessly, simply by typing the quantities and pressing RETURN. For example, typing ØA+2C (RETURN) will print up the answer 36, and BC−F1 will give CB. Only two-digit arithmetic can be carried out, but this is all that you normally need, because you can usually ignore the leading two bytes of addresses when you are performing hex arithmetic. You must, however, be rather careful when you are using this hex arithmetic to calculate branching displacements, because it does not allow for subtracting that extra 2.

Memory blocks

Oricmon has very useful actions that copy data from one part of memory to another, and which can compare data in two different parts of memory. Suppose that you have a set of bytes stored in addresses $8ØØØ to $8Ø5Ø, and you want them copied to a part of the memory that starts at address $5ØØØ. All you need to do is to type:

```
5ØØØ<8ØØØ.8Ø5ØM
```

and then press RETURN. The block will be copied into its new set of addresses ($5ØØØ to $5Ø5Ø), and the original block between $8ØØØ and $8Ø5Ø will remain in the memory. In this command (no spaces, remember), the letter 'M' is used to mean 'move'. If you use the same arrangement with the letter 'V', such as:

```
5ØØØ<8ØØØ.8Ø5ØV
```

then what happens is that the block of memory which starts at address $5ØØØ is compared, byte by byte, with the block from $8ØØØ to $8Ø5Ø. If any byte in the first set (the set that starts at &5ØØØ in this example) is not matched perfectly with the byte at the corresponding address in the other block, then the address and the size of the odd byte is displayed.

Disassembly

'Disassembly' is the opposite of assembly; it means the translation of machine code bytes into assembly language. For example, the disassembly of 8D Ø2 Ø4 would give STA $Ø4Ø2, which is a lot easier to understand. A disassembler is a very useful way of reading a machine code program. It's particularly useful if you want to understand how a machine code program works, and it's almost essential if you want to find out how other machine code programmers go about solving problems.

That said, very few disassemblers go all the way. Most simple disassemblers can't distinguish between machine code commands and ASCII text, for example, and there are very few disassemblers at any price or for any machine which will put in labels. The normal action when a branch command is encountered is to put in the address to which the jump is made. This is reasonably satisfactory if you have a printer, because you can inspect the printed copy later, and put in your own labels to show where the jumps go to. If you are tied to working on the screen, however, you will need to confine your disassembly to reasonably small chunks of machine code.

The Oricmon disassembly action is called up by typing L immediately following an address. As always, the address must be in hex, and there must be no space between the address and the letter L. When the RETURN key is pressed, fifteen lines of disassembly will be shown on the screen (or printer, if you have one). Each line shows the address of the first byte of the command, the hex codes, and the assembly language that corresponds to the codes. If you want another fifteen lines, you need only type another letter L, and press RETURN again. You can request multiples of fifteen lines (for a printer) by pressing the L key several times before you press RETURN.

If the machine code that you are disassembling contains text, then the disassembler will try to disassemble this as if it were code. Sooner or later, it will come across a byte which is not a valid 6502 code, and it will print a group of three question marks instead of a disassembly command. When you see ??? in a disassembly, you will need to check all the code around this point to find where the text begins and ends. You may have to repeat some disassembly from the address following the address at which the text ends. This is because the last text code may have been read as a command, making the disassembler read the next byte, a real instruction, as if it were data. If the disassembler gets started on the 'wrong foot' like this, it will eventually get back to normal, but you may get incorrect disassembly for several lines.

The assembler

In some ways, the assembler is the most useful part of any package of this type, because this is the part that allows you to create machine code. The Oricmon assembler, unlike some others, uses correct 6502 mnemonics, with the # sign indicating immediate loading, and all numbers in hex. Unfortunately, no labels can be used. As labelling is one of the most useful features of assembly language, this is a most unfortunate omission. It does not make the assember useless, because addresses can be used, but it makes the assembler slower in action, and more difficult to use. You have to note the address for each point that you have labelled. If you have a branch to a line that has not yet been written, you must fill in a 'guess' address, and then rewrite the line later when you have found the correct address. It's still a lot easier than assembling by hand, though!

The Oricmon assembler will assemble each line as code into memory when you press RETURN. This means that your 'source code', the lines of assembly language, cannot be recorded. This is another unfortunate omission, because it's a lot easier to record, replay and alter source code than assembler machine code. Each line of assembly language command has to start with a semicolon. If you are entering a long program, this can be tiring, so the Y key can be pressed to make the semicolon sign appear automatically in each line from then on.

You will want to specify the address at which assembly is to start. This has to be done *before* you press the Y key for automatic semicolons, so that you can type the start address, then a semicolon, and then the first assembly language command. You are limited as to the range of addresses that you can use. The Oricmon program is located in the memory from $8BØØ to $97FF, and it also makes use of addresses $4ØØ to $5ØØ. This means that you cannot assemble code into two of the most useful parts of memory which you would normally use! On some other machines, monitor programs can be shifted around in the memory so as to leave the bits that you want free, but this is not possible with Oricmon. The assemblers which are written in BASIC do at least allow both of these important parts of memory to be used. However, if you are developing machine code programs which are to be used instead of BASIC, rather than along with BASIC, you can assemble in the ordinary BASIC addresses from $5ØØ onwards.

As you press RETURN on each command line, the assembly language will be converted into code, and the code will be displayed. You must make a note of the address at which the code starts if you are going to use this address as a label. If you have entered an impossible instruction, the code will not appear, but an error message will! At the end of your program, you can press the DEL key to remove the semicolon, and the Y key to ensure that you don't get any more semicolons. Once this has been done, you are back into the ordinary monitor commands again. If the last command of your machine code program is an RTS byte, then the program will return to BASIC when it has finished. You may, however, want to check it while Oricmon is operating. If so, make the last byte that is executed a BRK instead of RTS. This will ensure that the program will return to Oricmon after running – if all goes well!

Execution hour!

Once you have assembled a program, you will want to record it. Oricmon has commands which will let you save and reload the code without returning to BASIC. This should *always* be done before you make any attempt to run a machine code program that you have assembled – we've gone into the reason in previous chapters. Once this has been done (remember that the **Slow** cassette speed is much more reliable for this type of save), you can try to run the program. You type the address of the start of the program, then the letter G. As usual, the address will be in hex, and there must be no space between the last character of the address and the letter G. When you press RETURN, the program will execute, and your moment of truth has come. If you ended the program with a BRK, then Oricmon will take over when your own program ends, and you can carry out a post-mortem!

Problem sorting

When a program returns to Oricmon, you will see a display on the screen. This will start with an address, which is the address number *following* the end of the program. The rest of the line shows the numbers that are stored in the 6502 registers. You can, in fact, find out what is stored in the registers at any time by typing the letter X (then RETURN). You can alter the contents of the registers by using address numbers $45 to $49. This can be useful if you want to check a loop. You can put a BRK byte at the end of the loop, place values into the 6502 registers, and then execute from the start of the loop. By placing suitable values in the registers, you can find if the loop will execute correctly, but without having to trace it round each time. This is a useful action, and one which is not all that common on monitors, certainly not on Oric/Atmos monitors.

Another extremely useful action is tracing. This allows you to run a machine code program in slow motion, either automatically or manually controlled. If you enter the start address followed by the letter T, the program will run in slow motion. Each instruction will be executed, and the 6502 register contents will be printed out before the next instruction is carried out. A byte stored in $68 controls the speed at which this tracing is done. You can stop the action by typing the backslash (\) character. If you find that this action is still faster than you want, you can type the starting address and the letter W. This will cause one instruction to be executed, and the register contents displayed. Another address and W are needed to carry out another instruction. The important point about both of these commands is that you do not necessarily have to start at the first command in the program. You *must* start at a command byte rather than a data byte, but this can be anywhere in the program. You can also alter the 6502 registers before you start tracing, and this, as I have indicated, allows loops to be executed in slow motion, and with a choice of values in registers.

All in all, the Oricmon offers a lot of facilities, and it was unfortunate that my copy was not adapted to the Atmos. It's likely, however, that by the time you read this, the adaptation will have been done, and you will be able to use this program. Despite the limitations of not being able to record source code, and of not permitting labels, Oricmon is useful, and a definite step in the right direction for the machine code programmer.

Chapter Nine
Last Round-up

One of the main problems about writing a book that introduces machine code programming is knowing where to stop. There is practically no limit to what can be added now that you started in this business, and where you go from here will depend a lot on what your particular interests are. I have to confess that my main interest is in 'utilities', the programs that make a machine more useful to me. Any utility program should make a machine easier to use, perhaps by adding a new command to the BASIC. Perhaps it would be a good idea to take a look at some ideas for utility programs, and at the same time learn some more about the operating system of the Oric/Atmos.

Have you ever typed NEW and pressed RETURN, then suddenly realised that you had not recorded the program? The BBC Micro has a BASIC command word OLD that lets you get your program back after an error like that, provided you have not entered another program after using NEW. What about giving your Oric/Atmos this capability? As you will know from the remarks that I made in Chapter 6, using NEW does not wipe out all the bytes of a program, it only changes the first two bytes to zeros. These are the bytes which would normally be used to find the address of the next line. If we can restore these two bytes, we can un-NEW a program! It's even easier than it seems, because the second of these bytes will be Ø5, and we only need to find what the first one will be. This assumes that your BASIC program has started at $Ø5ØØ as usual.

We should flowchart this problem, and Fig. 9.1 shows the result. If we start with a line number that is less than 256, then the byte in address $Ø5Ø4 will be zero, and the first byte of the program will be at address $Ø5Ø5. Now if we take each byte from then on, and check it, we will eventually come to a zero. This marks the end of that line. One byte beyond this point is the starting address for the next line. What we must do then, is to set a counter. It will have to start at 5,

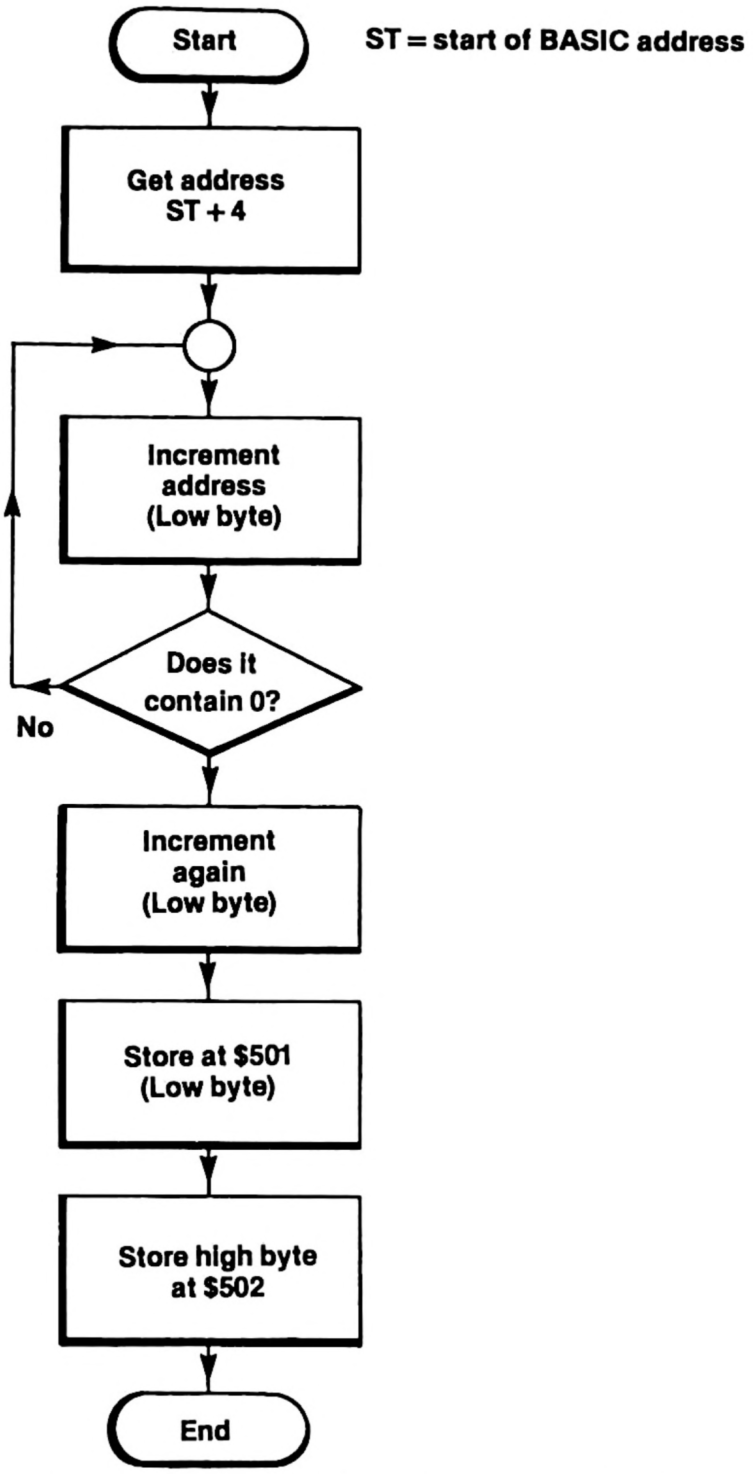


Fig. 9.1. The flowchart for the problem of recovering a program after using NEW.

and be incremented as we look at bytes from addresses $ø5ø5 on. When we find a zero, we increment the counter once more, and then place this number into address $ø5ø1, because it's the low-byte of the next line address. We then put the number $ø5 into address $ø5ø2, because that should be the high-byte of the next line address. That's all!

The next step is to put this into assembly language, and then into a BASIC program that will poke bytes into memory. We'll start in the way that we have been used to, as Fig. 9.2 shows. The X register is loaded with the number 4, and then incremented. The increment

(a)

```
            LDX #$Ø4          ;count number
MORE:       INX               ;incremented
            LDA $5ØØ,X        ;get code in first line
            BNE MORE          ;repeat if not zero
            INX               ;get start of next line
            STX $5Ø1          ;use as low byte
            LDA #$Ø5          ;high byte is Ø5
            STA $5Ø2          ;put it in place
            RTS               ;back to BASIC
```

(b)

```
10 HIMEM#95FF:A=#9600
20 FORN=0TO17:READ D$
30 POKEA+N,VAL("#"+D$):NEXT
100 DATAA2,4,E8,BD,0,5,D0,FA,E8,8E,1,
5,A9,5,8D,2,5,60
```

Fig. 9.2. (a) The assembly language and (b) the poke program for the 'un-new' action.

action comes at the start of a loop, so that the X register will be incremented on each pass through the loop. We then load the accumulator from $Ø5ØØ, X-indexed. This means that on the first time round, the accumulator will actually be loaded from $Ø5Ø5, because of the value of X. This is the first token byte of the program, and the first thing that we do is to test for this being Ø. If it isn't, we go back to the INX step, and repeat the indexed load. When we find a zero, the value of the number in X will be the count, from a starting number of $Ø5ØØ, of this address position. Adding 1 to X will give the count to the next line. This count number is then placed into address $Ø5Ø1, and the number 5 is placed in address $Ø5Ø2. For example, if we found that the first zero was found with X = 19, then the INX step makes X = 2Ø, and 2Ø is poked into $5Ø1, with 5 in $5Ø2, making the address $Ø52Ø, which should be the next line address. We should now be able to list and run the program that was NEWed. We do this by typing CALL $96ØØ. Try it!

The ! command

Though it's useful to be able to reverse the action of the NEW command by typing CALL $96ØØ, a snappier method would be better. As it happens, the Oric/Atmos provides for other ways of calling up machine code subroutines. One of these, USR, need not concern us here (for more information see Appendix E), but there are two more, ! and &. The & sign is one that needs to be followed by other information, enclosed in brackets, so we'll leave it out. The ! command looks more suitable for this job – but how does it work? To start with, you have to have your machine code stored in protected memory. The starting address of your machine code is then poked into two addresses, $Ø2F5 and $Ø2F6. When the ! sign is typed, and RETURN pressed, the machine will find your machine code program and run it. It sounds simple enough, so we'll give it a try.

We could, of course, just DOKE the starting address into place, but what I would prefer is to create a completely machine code routine which you could record, with autostart, so that you could keep it ready in the machine. This means that the start of the machine code must contain routines that place the correct addresses into memory, and then return to BASIC. Figure 9.3 shows what has to be added; it's only a few lines of load and store to get an address into $Ø2F5 and $Ø2F6. When the program is loaded, it will place the

(a)

```
                ORG $Ø4ØØ
LDR:            LDA #$ØB           ;lsb of start
                STA $Ø2F5;into !
                LDA #$Ø4           ;msb of start
                STA $Ø2F6;into !
     (un-NEW routine now put in starting at $Ø4ØB)
```

(b)

```
10 A=#400
20 FORN=0TO28:READ D$
30 POKEA+N,VAL("#"+D$):NEXT
40 CALL A
100 DATAA9,0B,8D,F5,2,A9,4,8D,F6,2,60
110 DATAA2,4,E8,BD,0,5,D0,FA,E8,8E,1,
5,A9,5,8D,2,5,60
```

Fig. 9.3. Adding lines to the 'un-new' program so that it can be called by pressing the ! key (then RETURN).

bytes into memory, and when it auto-runs, it will put its start address into $Ø2F5 and $Ø2F6. From then on, or until you use the reset button underneath the computer, pressing ! and RETURN will OLD your program, restoring anything that you have NEWed. Remember, though, that when a program has been NEWed, the VLT is set to an address near the start of BASIC. If you try to use a command like:

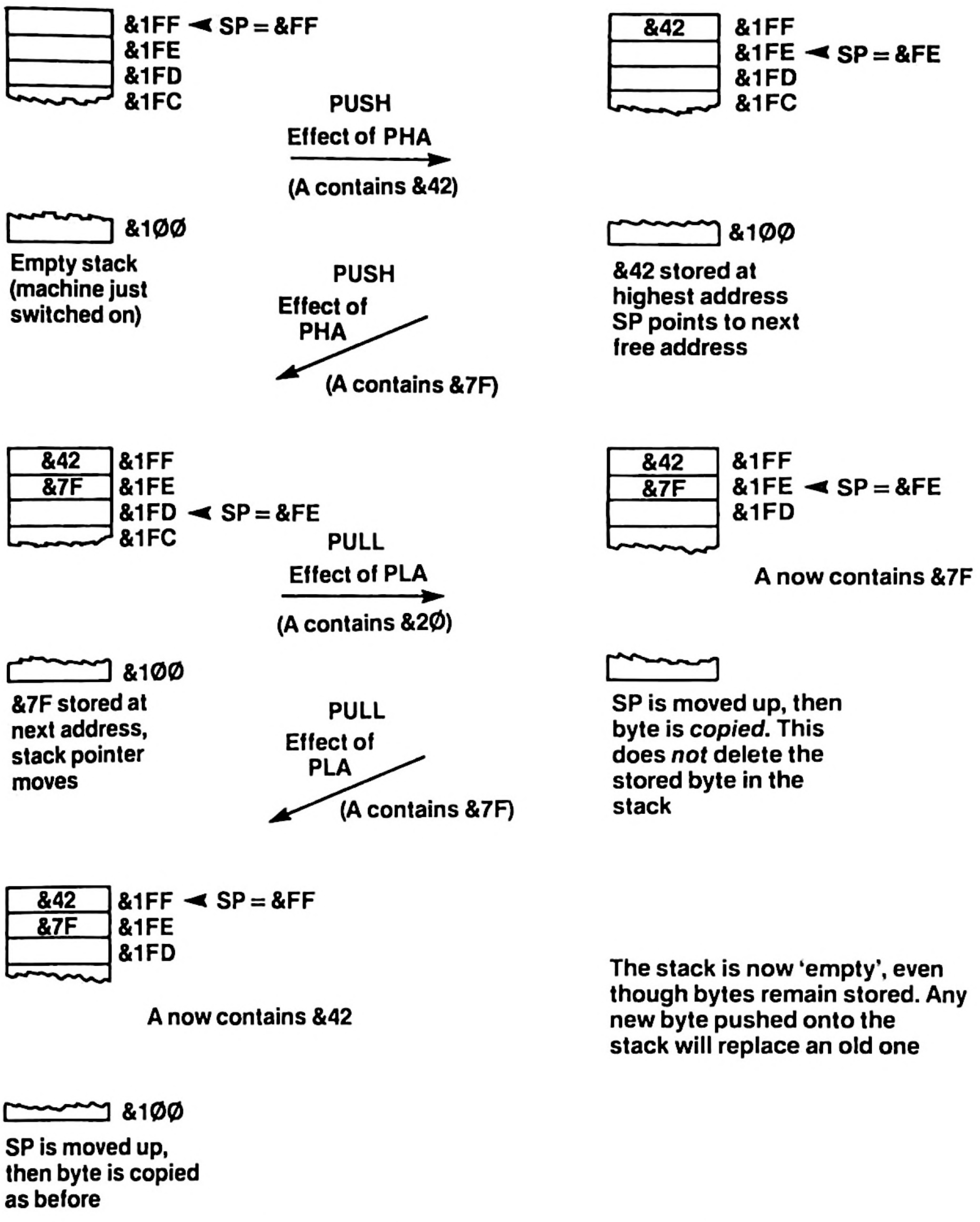
```
FOR N=Ø TO 1Ø:PRINTPEEK (#5ØØ+N); "";:NEXT
```

you will find that bytes of your program are replaced by VLT entries. This will usually stop the ! program from operating. The main novelty this time is that the program has been assembled to address $Ø4ØØ. The Oric/Atmos reserves addresses $Ø4ØØ to $Ø42Ø for your own short machine code routines. Anything that you assemble here will be untouched by any other machine action, whether BASIC or the action of the RESET button or anything else. Using the RESET button, however, will always reset the numbers in addresses $Ø2F5 and $Ø2F6.

The stack

You can't get much further in machine code programming without coming across the word *stack*. A stack is a section of memory, and its special use is to preserve bytes that have been kept in registers. There's no special set of memory chips that we use as a stack – but we do set aside part of the RAM for this purpose. For a 6502 microprocessor, we have to use memory in the address range $1ØØ to $1FF. What you probably find difficult to understand at the present time is why we should ever need to use memory in this way.

Let's take a simple example. Suppose you have a program in which the A register is being used to hold an ASCII code for a character. Suppose, now, in the middle of this program, that we want to create a time delay by making use of a count-down in the A register. Whenever we load the count value into the A register, we shall have replaced the ASCII code number that was stored there, and if we try to use the A register again in the rest of the program, we shall have to reload the code byte into it. This is what the stack is for. By means of a single byte instruction, we can store the contents of the accumulator or the status register in the stack memory, and by using another similar instruction, we can get the values back into the correct registers again. The act of storing the register(s) on the stack



&1FF ◄ SP=&FF
&1FE
&1FD
&1FC
&1ØØ
Empty stack (machine just switched on)
PUSH
Effect of PHA
(A contains &42)
&42
&1FF
&1FE ◄ SP=&FE
&1FD
&1FC
&1ØØ
&42 stored at highest address SP points to next free address
PUSH
Effect of PHA
(A contains &7F)
&42
&7F
&1FF
&1FE
&1FD ◄ SP=&FE
&1FC
&1ØØ
&7F stored at next address, stack pointer moves
PULL
Effect of PLA
(A contains &2Ø)
&42
&7F
&1FF
&1FE ◄ SP=&FE
&1FD
A now contains &7F
SP is moved up, then byte is *copied.* This does *not* delete the stored byte in the stack
PULL
Effect of PLA
(A contains &7F)
&42
&7F
&1FF ◄ SP=&FF
&1FE
&1FD
A now contains &42
&1ØØ
SP is moved up, then byte is copied as before
The stack is now 'empty', even though bytes remain stored. Any new byte pushed onto the stack will replace an old one

Fig. 9.4. The stack actions illustrated.

is called 'pushing', and recovering the values is called 'pulling'. These actions are illustrated in Fig. 9.4.

As a very simple example of the use of the stack, let's add another 'utility' to the Oric/Atmos. This time we'll add a program that will make your friends wonder if your Oric/Atmos is a special one, because you will be able to produce the 'shoot' sound just by typing *, instead of typing SHOOT and pressing RETURN. Yes, it's simple

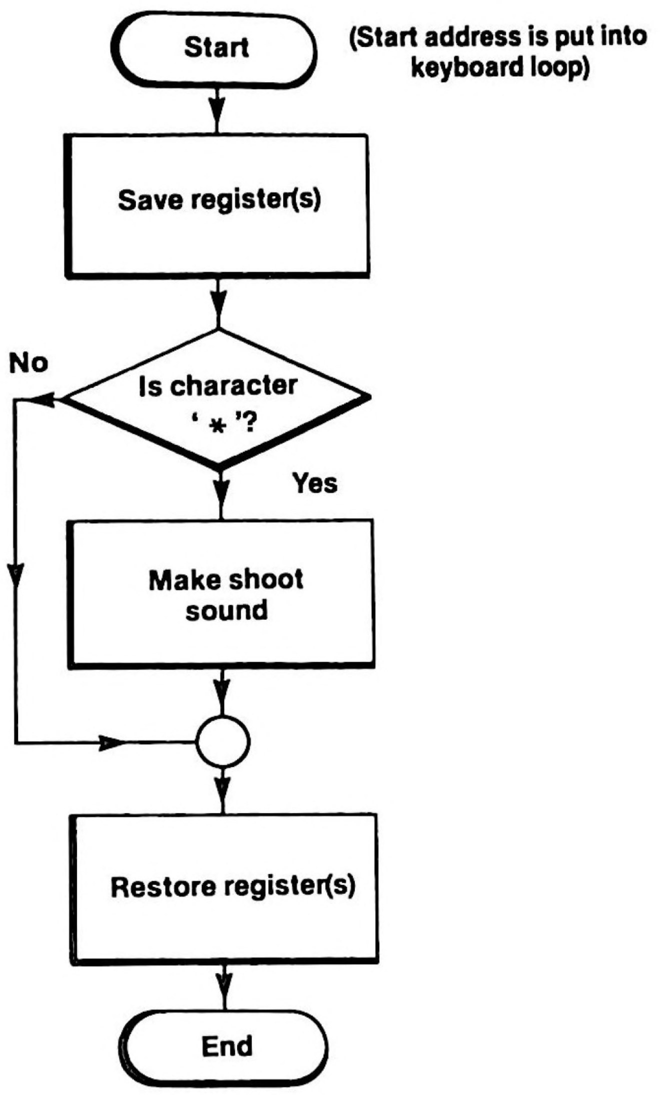


Fig. 9.5. A 'shoot' utility flowchart. The idea is that you get the SHOOT sound each time you press the asterisk key.

enough – when you know how! Figure 9.5 shows the flowchart. What we have to do is to intercept the keyboard routines. We already know how to do this, because we have done so in Chapter 6. We must then test the character in the accumulator. If this is the '*' character, ASCII code $1A, then we will carry out the 'SHOOT' routine. After this, or if the character is not '*', we go back to the normal routine.

Figure 9.6(a) shows the assembly language routine in detail. Addresses $Ø23C, $Ø23D contain the address of the GTORKB routine. Now the GTORKB routine is in ROM, and can't be changed, but addresses $Ø23C, $Ø23D are in RAM, and we most certainly can change them. Address $Ø23B holds a JMP instruction code, so that the whole group of three bytes is a sort of 'junction box', leading to the GTORKB routine. If we change the numbers in $Ø23C, $Ø23D to the start of a routine of our own, and end our

(a)

```
            ORG $Ø4ØØ
            LDA #$ØB           ;PATCH address lsb
            STA $Ø23C          ;into key routine addr.
            LDA #$Ø4           ;PATCH address msb
            STA $Ø23D          ;into key routine addr.
            RTS                ;back to BASIC
PATCH:      PHA                ;save accumulator
            CMP '*'            ;is it asterisk?
            BNE OUT            ;out if not
            JSR $FAB5          ;shoot routine
            PLA                ;restore accumulator
OUT:        JMP $EB78          ;back to key routine
```

Note: The Oric-1 may need a different JMP address.

(b)

```
10 A=#400
20 FORN=0TO22:READ D$
30 POKEA+N,VAL("#"+D$):NEXT
40 CALL A
100 DATAA9,0B,8D,3C,2,A9,4,8D,3D,2,60
,48,C9,2A,D0,3,20,B5,FA,68,4C,78,EB
```

Fig. 9.6. (a) The assembly language and (b) POKE lines for the asterisk-shoot program.

routine with a jump to the ROM address of GTORKB, then we shall have 'patched' our routine into the keyboard routine. This means that our routine will run each time GTORKB runs.

Now, as it happens, GTORKB runs continually while the machine is showing its READY prompt. Because of this, it wouldn't do to make the shoot sound occur each time GTORKB was used. By testing , however, we can make sure that only one character operates the SHOOT routine. The first line, however, is PHA. The character that is in the accumulator is going to be tested by the GTORKB routine, but if we call up the SHOOT routine, the accumulator will be used for something else, and we will have the wrong number in the accumulator when GTORKB runs. We avoid this by 'pushing the accumulator on the stack'. What the phrase means is that the number stored in the accumulator is put into the stack memory. Now we test the accumulator for the presence of $2A, the asterisk sign. If it's present, we continue with calling the shoot subroutine at $FAB5. If the character is not the asterisk, and also after the shoot routine, we 'pull the accumulator off the stack'. This means that we put back into the accumulator the number that was there before we started messing around with the SHOOT sound. We can then jump normally to the GTORKB routine itself.

Well, it works – after a fashion. When you type *, you hear the shoot sound effect, but things do not look right after that. For one thing, two asterisks in a row don't both produce the shoot sound. For another, peculiar things seem to happen to the display after the asterisk has been used, as you'll see when you LIST. What's wrong? The answer, almost every time, is that another of the registers is being corrupted. The SHOOT routine changes the values of all the registers, A, X, and Y. The effects that we see after using the program in Fig. 9.6 must be caused by the changes in the X or Y registers, perhaps both. The only way to be certain of not doing this is to save both of these registers on the stack.

It's at this point that the 6502 microprocessor shows its age. It does not have any commands for saving the X or Y registers on the stack! What you have to do is to pass the value in a register to the accumulator, and then save the accumulator on the stack. It's very long-winded, but that's all we can do. Let's design a Mark 2 version of the 'shoot-asterisk' program, in Fig. 9.7. This time, we'll test the accumulator right at the start of our routine. If the character in the accumulator is '*', then we'll go through all the push steps, but if it isn't, then we'll jump straight back to GTORKB. Only if the asterisk character is identified will we push the accumulator and the other two registers. We push the X register by the two steps TXA and PHA. This transfers the value in the X register into the accumulator, and then pushes this value on the stack. Similarly, TYA followed by PHA puts the value that was in the Y register on to the stack. By the time we get to the JSR $FAB5 step, all three registers have been saved on to the stack in this way.

After the SHOOT routine has run, we have to get the register contents back. This is where we have to be *very* careful. We have to 'unstack' in exactly the opposite order of stacking. The stack memory works on the 'last-in-first-out' principle. The last byte on the stack came from the Y register, so it's the first to be restored, using PLA (pull from stack to accumulator) and TAY (transfer A to Y). Then we get the X register byte back by using PLA and TAX. Finally, another PLA gets back the correct byte for the accumulator itself. Note the order – the accumulator is always the first saved and the last recovered. Try the routine, and you'll see that it's free of the troubles that affected the earlier version.

(a)

```
          LDA #$ØB                 ;load as before
          STA $Ø23C
          LDA #$Ø4
          STA $Ø23D
          RTS
PATCH:    CMP '*'                  ;is it * ?
          BNE OUT                  ;back if not
          PHA                      ;save A
          TXA                      ;put X in A
          PHA                      ;and save it
          TYA                      ;put Y in A
          PHA                      ;and save it
          JSR $FAB5                ;shoot routine
          PLA                      ;get back Y
          TAY                      ;put it in place
          PLA                      ;get back X
          TAX                      ;put it in place
          PLA                      ;get back A
OUT:      JMP $EB78                ;key routine
```

(b)

```
10 A=#400
20 FORN=0TO30:READ D$
30 POKEA+N,VAL("#"+D$):NEXT
40 CALL A
100 DATAA9,0B,8D,3C,2,A9,4,8D,3D,2,60
110 DATAC9,2A,D0,0D,48,8A,48,98,48,20
,B5,FA,68,A8,68,AA,68,4C,78,EB
```

Fig. 9.7. An improved asterisk-shoot program which preserves all of the registers.

Program protection

When you have spent a lot of time and effort on a program, you might want to prevent anyone from copying it while you were proudly showing it off on Club Night. A lot of time and effort goes into program protection these days, particularly in commercial programs, because without protection, it has been estimated that there will be ten illegal copies around for each one that has been bought. This is a sad situation, and there's no simple way round it.

Anyone who has spent a lot of time on developing a program that will be sold for cash will naturally want to make as much as possible out of it while there is an interest. A lot of commercial programs, particularly games, sell in huge numbers for a few weeks, and after that not at all. This is due to two factors – the arrival of a new game which everybody wants, and the existence of thousands of pirated copies of the old one.

The other side of this problem is that software on cassettes is not very reliable, and even disks can crash. I would never use a program for which I did not have a back-up copy. In addition, I often want to alter a program to suit my own needs, perhaps by making a poor colour display into black/white, or by adding a printer routine in place of a screen display. If a program cannot be listed or copied, then I can't do these things.

Until something can be worked out (and it will take a lot to make people refuse to buy pirated software) you may want to protect your programs. I think that it's important to emphasise that no program can ever be absolutely 100% copy-proof. You can, for example, buy twin-cassette recorders. You pop your program cassette in one side, a blank in the other, press the button and make a copy. When you record your Oric/Atmos programs at the fast speed, though, these copiers do not work reliably. One up for protection!

Since you can't protect a program against anyone who is knowledgeable (like a good machine code programmer!) all that you can hope for is to protect against the kind of guy who borrows your tapes, makes a quick copy and gives the tapes back. This sort of copier won't spend any time in trying to figure a way round protection, he will just give up. Here are some tips.

(1) Make your program auto-start, and make the last command NEW. This will ensure that when the program ends, there is nothing to list – unless the user has my ! program in the memory.

(2) Disable the stop (CTRL C) and RESET actions. This is most easily done by using POKE #1A,1 at the start of your program, and POKE #1A,76 where the program should normally end. If CTRL C or the RESET button are used while address #1A contains 1, then the operating system will go into a loop with an illegal quantity error message scrolling. The effect of this poke is to disable a jump which takes place to address $CCBØ when a BASIC program ends.
(3) By doking a new address into #1B, #1C, you can make a jump to any other address when a BASIC program is stopped. This requires

more work, but it could allow you to run a machine code program which printed the message:

COPY PROTECTED – HARD LUCK

and disabled the keyboard.

(4) Instead of using NEW at the program end, and a disabling step if the program is stopped, run a routine that completely clears the memory! By using JSR $C000 (or $ECCC) you will run the routines that the Oric/Atmos runs when it is switched on at first, ensuring that nothing remains in the memory.

If you posses an Atmos, however, all of this work will be lost if a copy-artist loads your program from a tape recorder that is not well matched to an Atmos. The error-detecting routine of the Atmos (not used in Oric-1) often works so well that it detects errors that aren't there, and this will prevent the auto-start. The copier can then examine your program, and remove the POKE lines that are the usual dead give-away. It's possible to prevent this by using a short machine code program to load in the BASIC – but that's getting into advanced machine code work, well beyond the scope of this book. If this subject interests you, then keep your eyes on the Oric magazines for hints, and keep in touch with the user group. This is where all the useful information comes from.

The renumber problem

Another useful facility that the Oric/Atmos lacks is a renumbering routine. Now a *real* renumbering routine is quite a piece of art. It will renumber all the lines of a BASIC program for you, taking any starting number (within reason) and any line increment (again within reason) that you like to use. Not only are the lines themselves renumbered, but also the line numbers in GOTO and GOSUB statements. A listing for a machine code renumber program can be several feet long, and that's not what we are attempting. What I want to do here is to show how to start about a renumbering exercise, because this is something that can be very useful for a lot of other things. As always, it's better to aim at something simple to start with. For that reason, we'll try to renumber, with a starting number that must be less than 255, and with line increments of 1. We'll also restrict the number of lines that can be renumbered, so that no line number must exceed 255 after renumbering. If that sounds modest, remember that it allows you to renumber programs of up to 255 lines. It also makes your renumbered programs very difficult for anyone else to alter, because they can't squeeze another line in anywhere if the lines are numbered consecutively.

(a)

```
(Start number in address $Ø7)

         LDA $9A            ;get start lsb
         STA $Ø5            ;put it in $Ø5
         LDA $9B            ;get start msb
         STA $Ø6            ;put it in $Ø6
LOOP:    LDY #1             ;index
         LDA ($Ø5),Y        ;get next line high
         BEQ OUT            ;if Ø, end of program
         INY                ;bump up index
         LDA $Ø7            ;get start number
         STA ($Ø5),Y        ;put in new number lsb
         INY                ;bump index
         INC $Ø7            ;bump number also
         LDA #$ØØ           ;for line number high
         STA($Ø5),Y         ;put it in place
         TAY                ;zero Y register also
         LDA ($Ø5),Y        ;next line lsb
         STA $Ø8            ;put it in Ø8
         INY                ;bump index
         LDA ($05),Y        ;next line msb
         STA $Ø9            ;put it in Ø9
         LDA $Ø8            ;get lsb
         STA $Ø5            ;put in current line
         LDA $Ø9            ;get msb
         STA $Ø6            ;put in current line
         BPL LOOP           ;repeat actions
OUT:     RTS                ;back to BASIC
```

(b)

```
 10 HIMEM#95FF:A=#9600
 20 FORN=0TO46:READ D$
 30 POKE A+N,VAL("#"+D$):NEXT
 100 DATAA5,9A,85,5,A5,9B,85,6,A0,1,B1
,5,F0,20,C8,A5,7,91,5,C8,E6,7,A9,0
 110 DATA91,5,A8,B1,5,85,8,C8,B1,5,85,
9,A5,8,85,5,A5;9,85,6,10,DA,60
```

Fig. 9.8. (a) The assembly language listing for the simple renumber program, and (b) the BASIC listing to poke the bytes into place.

The flowchart for this exercise is illustrated in Fig. 9.8(a). The idea is simple enough. The address for the start of BASIC is transferred from addresses $9A, $9B, into two other zero page addresses, $ø5, $ø6. I originally chose $25, $26, but found that these were being used for other things. This is something that you have to be very careful about when you make use of zero page addresses. It's always a good idea to check them while you are testing a program, because the Oric/Atmos manual does not document their uses fully.

Having transferred the address for the start of the first line of BASIC, the program now starts a loop. The Y register is loaded with 1, and the indirect load LDA ($ø5),Y gets the byte from address $5ø1+1 = $5ø2. This will be the high-byte of the next-line address. When this is zero, we need to end the program (end of program marker), and the BEQ OUT step will do this. If it is not zero, then we can continue the renumbering procedure. The Y register is incremented, so that the number 2 is stored. The accumulator is loaded from address $ø7, which holds the starting line number. Conventionally, you might use 1ø here. This number is placed into address 5ø3 by the indirect store command STA ($ø5),Y. The Y register is then incremented again, and the address $ø7 is incremented to give the next line number, 11 in this case.

The accumulator is loaded with zero, and this is stored at address $5ø4. This is the upper byte of line number, and in this simple program, we are not using this byte. Setting it to zero means that if you have line numbers greater than 255 in the original program, they will be correctly reset to low numbers. The zero in the accumulator is then transferred to the Y register by the TAY command, and the accumulator is loaded from address $5ø1. This will be the low-byte of the next-line address, and it is stored at address $ø8. The Y register is incremented so that the accumulator can be loaded from $5ø2. This gets the high-byte of the next line address, and it is stored into address $ø9. What we now want to do is to put the next line address numbers into the addresses $ø5 and $ø6 that we used for the start address of the first line. If we can do this, we can then repeat all the steps of the loop.

This is done by a set of load and store operations. The last of these will have used a positive or zero number in the accumulator (certainly not negative), and we can test this to cause the loop. Once again, I could have used JMP, with a complete address, but BPL is neater. It means 'branch if positive or zero', and it only need be followed by a single byte displacement. The reason I prefer to use branch commands like this rather than JMP is that branch commands make the program 'relocatable'. A relocatable program can be put anywhere in memory, not just into a certain range of addresses. If you assemble a program at address $96ØØ, and it contains somewhere a step like JMP $9654, then you are stuck with that. If you decide that you want to place the program starting at address $7ØØØ, then the JMP $9654 step will have to be changed to $7Ø54. If you have no jumps to locations within the program (you can have jumps to ROM or to zero page if you like), then you can shift the bytes of a program anywhere in memory and still expect it to work. That's a great advantage, especially if you are using POKE statements to load your machine code.

Getting back to business, then. You run the BASIC program of Fig. 9.8(b) which puts the bytes in place. There's no CALL, because you might not want to renumber this program. You then load in the program that you want to renumber. You poke into address $Ø7 the starting line number, usually 1 or 1Ø. If you forget, the first line number will be Ø. Then use CALL $96ØØ, and your program will be renumbered in steps of 1 – instantly!

The way ahead

No program is ever complete, it's just a stepping stone to the next one. The renumber routine is a good example of a simple beginning that can be the foundation for something a lot larger. To start with, we could fairly easily make use of the high-byte of the line number, so that when the low-byte of the line number exceeded 255, the high-byte was incremented. We will have to keep a starting value of high-byte in zero page memory, just as we keep the low-byte. Adding this allows you to use a much greater range of line numbers, and to number in tens if you like. The program at this stage, however, is getting to a length which makes you want to use an assembler. Life is too short to work on machine code programs of this length with POKE methods. Apart from anything else, it's too wearing to alter all the DATA lines each time you think of something new to add.

Rewriting the GOTO and GOSUB lines is something quite different, and though it's well beyond the scope of this book, it's as well to see why. It's not difficult to understand how your program can be adapted to look through a line, and jump to a subroutine when it finds a GOTO, GOSUB or THEN. Each of these is represented by a token, and you only have to add a bit of indirect addressing, and a few CMP steps to get this far. The trouble starts after this. When you have, in BASIC, a line which ends with GOTO 12ØØ, for example, the 12ØØ causes problems. This number is stored as four ASCII codes, not as a two-byte code. The ROM of the Oric/Atmos must contain a routine which will read a set of ASCII codes for numbers, and convert into two-byte form, but you have to be sure that you read the correct number of codes, with no letter codes. Having got this number you then have to find the address of the line that it refers to. This line may not yet have been renumbered, so you have to store the address of the GOTO line, and the address at which the line number is placed. When the line is renumbered, its number has to be converted back to ASCII codes, and these codes placed into the correct part of memory. It's not easy – particularly when you consider that a GOTO15 might become a GOTO15ØØ, requiring more memory space, and that there might be thirty of them in a program.

We have come to the end of this particular road now. You now know how to write a machine code program, how to place it in memory, and how to run it. What you need now, more than anything else, is experience. You need to study every machine code program that you can lay your hands on. Some of them may not be very useful. There are, for example, a number of programs that work on the Oric-1 which will not work on the Atmos because of changes in the ROM. Nevertheless, any machine code program for the 6502 microprocessor will have something of interest for you. The 6502 is used in the BBC Micro and in the Commodore 64, and a lot of machine code programs are printed for these two machines. Once you can see what a machine code program is trying to do, you can adapt it for the Oric-1 or the Atmos. For the rest, it's learning addresses and routines all the time. Keep in touch with your user group, magazines such as *Oric Owner* and *Oric Computing*, and the information that you need will flow in your direction.

Appendix A
How Numbers are Stored

The Oric/Atmos uses five bytes of memory to store any number. This Appendix describes how numbers are stored, but if you have no head for mathematics, you may not be any the wiser!

To start with, floating point (not integer) numbers are stored in *mantissa-exponent* form. This is a form that is also used for denary numbers. For example, we can write the number 216ØØØ as 2.16 × 10^5, or the number .ØØØ12 as 1.2 × 10^{-4}. When this form of writing numbers is used, the power (of ten in this case) is called the *exponent*, and the multiplier (a number greater than Ø and less than 1) is called the *mantissa*. Binary numbers can also be written in this way, but with some differences. To start with, the mantissa of a binary number that is written in this form is always fractional, but no point is written. Secondly, the exponent is a power of two rather than a power of ten. We could therefore write the binary number 1Ø11ØØØØ as 1Ø11E1ØØØ. This means a mantissa of 1Ø11 (imagine it as .1Ø11) and exponent of 1ØØØ (2 to the power 8 in denary). There's no advantage in writing small numbers in this way, but for large numbers, it's a considerable advantage. The number:

11Ø1Ø1ØØØØØØØØØØØØØØØØØØ

for example can be written as 11Ø1Ø1E11ØØØ (think of it as .11Ø1Ø1 × 2^{24}).

This scheme is adapted for the Oric/Atmos, and other machines which use Microsoft BASIC. Since the most significant digit of the mantissa (the fractional part of the number) is always 1 when a number is converted to this form, it is converted to a Ø for storage purposes. The exponent then has (denary) 128 added to it before being stored. This allows numbers with negative exponents up to −128 to be stored without complications, since a negative exponent is then stored as a number whose value is less than 128 denary. The

Oric/Atmos uses four bytes to store the mantissa of a number, and one byte to store the exponent.

To take a simple example, consider how the number 2∅ (denary) would be coded. This converts to binary as 1∅1∅∅, which is .1∅1∅∅∅∅∅ × 2^5, writing it with the binary point shown, using eight bits, and with the exponent in denary form. The msb of the fraction is then changed to ∅, so that the number stored is ∅∅1∅∅∅∅∅. Peeking this memory will therefore produce the number (denary) 32 in the mantissa lowest byte. Meantime, the exponent of 5 is in binary 1∅1. Denary 128 is added to this, to make 1∅∅∅∅1∅1. Peeking this memory will give you 133 (which is 128+5).

Integers need only two bytes for storage. The integer must have a value between −32768 and +32767. To convert an integer to the form in which the Oric/Atmos stores it, proceed as follows:

1. If the number is negative, subtract it from 65536 and use the result.
2. Divide the number by 256 and take the whole part. This is the most significant byte.
3. Subtract from the number 256 × (most significant byte). This gives the least significant byte.

Example: Convert 9438 into integer storage form.

The number is positive, so it can be used directly.
9438/256=36.867187.
The msb is therefore 36.
36*256=9216, and 9438−9216=222.
The low byte is 222.

Appendix B

Hex and Denary Conversions

(a) Hex to Denary

For single bytes (two hex digits) -
Multiply the most significant digit by 16, and add the other digit.
For example:

$3D is 3*16 + 13 =61 denary.

For double bytes (address numbers) -
Write down the least significant digit. Now write under it the value of the next digit, multiplied by 16. Under that, write the next digit, multiplied by 256. Under that, write the next digit, multiplied by 4ø96.
For example:

$F3DB converts as follows:

Write 1s digit	11
Next digit*16 is 13*16	2ø8
Next digit*256 is 3*256	768
Next digit*4ø96 is 15*4ø96	6144ø
Now take the total, which is	62427

Denary to hex

For single bytes (less than 256 denary) -
Divide by 16. The whole part of the number is the most significant digit. The least significant digit is the fractional part of the result multiplied by 16.

For example:

To convert 155 to hex:
153/16 = 9.6875, so 9 is the most significant digit.
The least significant digit is .6875*16, which is 11. This converts to hex B, so that the number is $9B.

For double-byte numbers (numbers between 256 and 65535 denary)
Divide by 16 as before. Note the whole number part of the result, and write down the fractional part, times 16, as a hex digit. Repeat the action with the whole number part, until only a single hex digit remains.
For example:

To convert 23815 to hex:
23815/16=1488.4375. The fraction .4375*16 gives 7, and this is the least significant digit.
Taking the whole number part, 1488/16=93.ØØ. Since there is no fraction, the next hex digit is Ø.
93/16=5.8125. The fraction .8125, multiplied by 16 gives 13, which is hex D. This is the third hex figure. Since the whole number part is less than 16 (it's 5), then this is the most significant digit, and the whole number is $5DØ7.

Appendix C

The Instruction Set

The instruction set of the 6502 is a relatively small one, but there will be some instructions in this list which you may never need to use unless you go in for very advanced programming indeed. A full description of the action of each instruction would take too much space, and so the action has been indicated by abbreviations. For a full description, see one of the books devoted to 6502 programming. In general, M means a byte at an address in memory, and the registers are referred to under their usual letter references. An arrow indicates where the result of an action is stored. For example, $A+M+C \rightarrow A+C$ means that the byte in the memory (addressed by the instruction) is added to the byte in the accumulator A, plus the carry C, and the result is placed in the accumulator A, with another possible carry in C.

The instruction codes are shown in columns graded by the addressing method. These methods are Immediate, Zero Page, Zero Page X Indexed, Absolute, Absolute X Indexed, Absolute Y Indexed, Indirect X, Indirect Y, and PC Relative. A few instructions use Implied Addressing. The implied addressing method means that no special addressing is needed. In the assembly language forms, ADDR means a full two-byte address and addr means a single (lower) byte address for zero page addressing. Disp is used to mean displacement in PC Relative addressing. Byte means the byte following an immediate addresses code. S has been used to mean the Processor Status register. Flags are referred to as C,N and V. All op codes are shown in hex.

Assembly language form	*Addressing method*	*Opcode*	*Action*
ADC # Byte	Immediate	69	A+M+C→A+C
ADC addr	Zero page	65	
ADC addr, X	Zero page, X	75	
ADC ADDR	Absolute	6D	
ADC ADDR,X	Absolute, X	7D	
ADC ADDR, Y	Absolute, Y	79	
ADC (addr, X)	Indirect, X	61	
ADC (addr), Y	Indirect, Y	71	
AND # Byte	Immediate	29	A AND M→A
AND addr	Zero page	25	
AND addr, X	Zero page, X	35	
AND ADDR	Absolute	2D	
AND ADDR,X	Absolute, X	3D	
AND ADDR, Y	Absolute, Y	39	
AND (addr, X)	Indirect, X	21	
AND (addr), Y	Indirect, Y	31	
ASL A	Implied	∅A	Shift left
ASL addr	Zero page	∅6	
ASL addr, X	Zero page, X	16	
ASL ADDR	Absolute	∅E	
ASL ADDR, X	Absolute, X	1E	
BCC Disp	Relative	9∅	Branch if C=∅
BCS Disp	Relative	B∅	Branch if C=1
BEQ Disp	Relative	F∅	Branch if Z=1
BIT addr	Zero page	24	OR with M,
BIT ADDR	Absolute	2C	Test N & V
BMI Disp	Relative	3∅	Branch if N=1
BNE Disp	Relative	D∅	Branch if Z=∅
BPL Disp	Relative	1∅	Branch if N=∅
BRK	Implied	∅∅	Interrupt program
BVC Disp	Relative	5∅	Branch if V=∅
BVS Disp	Relative	7∅	Branch if V=1
CLC	Implied	18	Clear carry
CLD	Implied	D8	Clear decimal mode

Assembly language form	*Addressing method*	*Opcode*	*Action*
CLI	Implied	58	Clear interrupt disable
CLV	Implied	B8	Clear V flag
CMP #Byte	Immediate	C9	A-M, set flags
CMP addr	Zero page	C5	
CMP addr, X	Zero page, X	D5	
CMP ADDR	Absolute	CD	
CMP ADDR, X	Absolute, X	DD	
CMP ADDR, Y	Absolute, Y	D9	
CMP (addr, X)	Indirect, X	C1	
CMP (addr), Y	Indirect, Y	D1	
CPX # Byte	Immediate	E∅	X-M set flags
CPX addr	Zero page	E4	
CPX ADDR	Absolute	EC	
CPY # Byte	Immediate	C∅	Y-M set flags
CPY addr	Zero page	C4	
CPY ADDR	Absolute	CC	
DEC addr	Zero page	C6	M-1→M
DEC addr, X	Zero page, X	D6	
DEC ADDR	Absolute	CE	
DEC ADDR, X	Absolute, X	DE	
DEX	Implied	CA	X-1→X
DEY	Implied	88	Y-1→Y
EOR # Byte	Immediate	49	A EOR M>A
EOR addr	Zero page	45	
EOR addr, X	Zero page, X	55	
EOR ADDR	Absolute	4D	
EOR ADDR, X	Absolute, X	5D	
EOR ADDR, Y	Absolute, Y	59	
EOR (addr, X)	Indirect, X	41	
EOR (addr), Y	Indirect, Y	51	
INC addr	Zero page	E6	M+1→M
INC addr, X	Zero page, X	F6	
INC ADDR	Absolute	EE	
INC ADDR, X	Absolute, X	FE	
INX	Implied	E8	X+1→X
INY	Implied	C8	Y+1→Y

Assembly language form	*Addressing method*	*Opcode*	*Action*
JMP ADDR	Absolute	4C	Jump to ADDR
JMP (ADDR)	Indirect	6C	Jump to stored address
JSR ADDR	Absolute	20	Jump to subroutine
LDA # Byte	Immediate	A9	M→A
LDA addr	Zero page	A5	
LDA addr, X	Zero page, X	B5	
LDA ADDR	Absolute	AD	
LDA ADDR, X	Absolute, X	BD	
LDA ADDR, Y	Absolute, Y	B9	
LDA (ADDR, X)	Indirect, X	A1	
LDA (ADDR), Y	Indirect, Y	B1	
LDX # Byte	Immediate	A2	M→X
LDX addr	Zero page	A6	
LDX addr, Y	Zero page, Y	B6	
LDX ADDR	Absolute	AE	
LDX ADDR, Y	Absolute, Y	BE	
LDY # Byte	Immediate	AØ	M→Y
LDY addr	Zero page	A4	
LDY addr, X	Zero page, X	B4	
LDY ADDR	Absolute	AC	
LDY ADDR, X	Absolute, X	BC	
LSR A	Implied	4A	Shift right
LSR addr	Zero page	46	
LSR addr, X	Zero page, X	56	
LSR ADDR	Absolute	4E	
LSR ADDR, X	Absolute, X	5E	
NOP	Implied	EA	No operation
ORA # Byte	Immediate	Ø9	A OR M→A
ORA addr	Zero page	Ø5	
ORA addr, X	Zero page, X	15	
ORA ADDR	Absolute	ØD	
ORA ADDR, X	Absolute, X	1D	
ORA ADDR, Y	Absolute, Y	19	
ORA (addr, X)	Indirect, X	Ø1	
ORA (addr), Y	Indirect, Y	11	
PHA	Implied	48	Push A on stack

Assembly language form	*Addressing method*	*Opcode*	*Action*
PHP	Implied	∅8	Push S on stack
PLA	Implied	68	Pull A from stack
PLP	Implied	28	Pull S from stack
ROL A	Implied	2A	Rotate left
ROL addr	Zero page	26	
ROL addr, X	Zero page, X	36	
ROL ADDR	Absolute	2E	
ROL ADDR, X	Absolute, X	3E	
ROR A	Implied	6A	Rotate right
ROR addr	Zero page	66	
ROR addr, X	Zero page, X	76	
ROR ADDR	Absolute	6E	
ROR ADDR, X	Absolute, X	7E	
RTI	Implied	4∅	Return from interrupt
RTS	Implied	6∅	Return from sub-Routine
SBC # Byte	Immediate	E9	A-M-C*→M
SBC addr	Zero page	E5	C* is a borrow
SBC addr, X	Zero page, X	F5	
SBC ADDR	Absolute	ED	
SBC ADDR, X	Absolute, X	FD	
SBC ADDR, Y	Absolute, Y	F9	
SBC (addr, X)	Indirect, X	E1	
SBC (addr), Y	Indirect, Y	F1	
SEC	Implied	38	Set carry flag
SED	Implied	F8	Set decimal mode
SEI	Implied	78	Set interrupt disable
STA addr	Zero page	85	A→M
STA addr, X	Zero page, X	95	
STA ADDR	Absolute	8D	
STA ADDR, X	Absolute, X	9D	
STA ADDR, Y	Absolute, Y	99	
STA (addr, X)	Indirect, X	81	
STA (addr), Y	Indirect, Y	91	
STX addr	Zero page	86	X→M
STX addr, Y	Zero page, Y	96	
STX ADDR	Absolute	8E	

Assembly language form	*Addressing method*	*Opcode*	*Action*
STY addr	Zero page	84	Y→M
STY addr, X	Zero page, X	94	
STY ADDR	Absolute	8C	
TAX	Implied	AA	A→X
TAY	Implied	A8	A→Y
TYA	Implied	98	Y→A
TSX	Implied	BA	S→X
TXA	Implied	8A	X→A
TXS	Implied	9A	X→S

Appendix D

Addressing Methods of the 6502

Each addressing method has the effect of using a byte in the memory. The address at which this byte is stored is called the Effective Address (EA). The purpose of any addressing method is to make use of an effective address.

Immediate Addressing: The EA is the address that immediately follows the instruction byte.

Zero Page Addressing: Only the lower byte of the EA is given in the instruction. The upper byte is always ØØ, hence the name of zero page. For example, using zero page addressing with a byte of FB would make use of the address $ØØFB.

Absolute Addressing: The instruction is followed by two bytes, which form a complete address. For example, LDA $563F means that the accumulator is to be loaded from the address $563F.

Indexed Addressing: A number is stored in one of the index registers (X or Y). The effective address is this number plus any address that is specified in the instruction. For example, LDA 25, X means load the accumulator from the address which consists of the number stored in the X register, plus 25.

Implied Addressing: The address is implied by the instruction, and no special address reference is needed. For example, INX means increment the X register, and no EA is needed.

Indirect Addressing: There are two forms, both of which use zero page addresses. The X-indexed indirect adds the contents of the X register to the zero page address number, and fetches the byte at this address. This forms the low byte of the effective address. The high byte is then fetched from the next higher page zero address. The Y-indexed indirect fetches the byte from the page zero address, and then adds the contents of the Y register to this byte. This is the lower byte of the effective address, and the higher byte is fetched from the next zero page address as before.

Appendix E
Passing Parameters

Since this book has been intended as a gentle introduction, I have tried as far as possible to duck under some of the more difficult problems that can arise when you want to program in machine code. One of these problems is that of 'passing parameters', and it's particularly important when you use a combination of BASIC and machine code for programming, as many programmers do.

First of all, what is meant by 'passing parameters'? A parameter is a quantity that has a value which may be fixed for a time but which can also be altered. In BASIC, we often use variables as parameters, meaning that they take values for a short time only. When you program a loop that starts FOR N=1 TO 5, for example, the value of X will be 1 on the first pass through the loop, 2 in the next pass, and so on. Between the FOR line and the NEXT line, the value of X is fixed – it doesn't change until the loop is completed. That's what we mean by a parameter keeping its value for a time.

When you get into more advanced programming, you will often encounter the problem of 'passing parameters'. This usually means making use of a parameter in more than one routine. For example, suppose you had a BASIC program which in line 1ØØØ had a subroutine for printing text centred on the screen. In this subroutine, the text might be represented by a variable name of TX$. Now if you happen to have a title which is assigned to the name of TX$, you can print it centred by having the instruction GOSUB1ØØØ in your program. What do you do if your heading happens to use the variable name of NM$? Simple, in BASIC, you just use the line:

```
    TX$=NM$:GOSUB1ØØØ
```

The TX$=NM$ part *passes the value of parameter NM$ to the name TX$*, and this is what passing parameters, at its simplest, is about

This book is about machine-code programming, however, and the problem that is likely to haunt us is how to pass parameters to a machine code program from a BASIC program. As you might expect, there is more than one possible method, and the simplest is to use memory as what is called a 'parameter block'. A parameter block is a set of numbers stored in consecutive addresses in memory, which the machine code program can make use of. The parameter block can be filled by poking numbers from a BASIC program, or by storing them from another machine-code program. It's even possible to have one program store numbers which another program will use later.

Take a simple example. Suppose you have a BASIC program that requires you to input a title. The machine code program will animate this title, making the letters appear to fly all over the screen. We're not going to look at the whole program, just the problem of parameter passing. What has to be passed is a set of ASCII codes, the codes for the letters of the message. We can use a pair of lines like:

```
500 AD=38800:FOR N=1 TO LEN(A$)
510 POKE AD+N,ASC(MID$(A$,N,1)):NEXT
```

which will place the ASCII codes into memory starting at address 38801. This is our parameter block, and you have to ensure that you have some method of reading the correct number of bytes when the machine-code runs. You could, for example, have a terminator after the last ASCII code, or you could place the count number, LEN(A$) into another memory address (perhaps 38800) so that the machine-code program could read it.

A parameter block is a useful way of passing quite a large amount of information from a BASIC program to machine code, and it can also be used to pass information in the other direction. When a single number has to be passed, however, there is a rather more convenient method, which makes use of the USR command. Just as you have to specify the address of a machine code routine to use with CALL, you also need to specify an address with USR. This is done by a line in your BASIC program which reads:

```
100 DEFUSR=38400
```

– using whatever address number you have allocated for the start of your machine code. The difference between this and CALL38400, however, is that when this line is run, the machine code program *isn't* called! All that happens is that the address is stored in memory *ready* for a call. To call the machine code into action, you must use a line like:

```
PRINT USR(24.5) or A = USR(4237)
```

The number within the brackets is the parameter which will be passed to the machine code program, and it could be a variable name, as in A=USR(X).

The next thing to look at is what happens to this number. It is stored in what Oric call the 'floating-point accumulator'. This is a parameter block in the memory, which takes addresses $D∅ to $D5. To understand how numbers are stored here, you will have to grapple with Appendix A again. The format is slightly different. Address $D∅ is used to store the exponent of the number (in binary fraction form), after adding 128 to the value. The next four bytes, from $D1 to $D4 are used to store the mantissa. Address $D5 is used to store a sign byte, with ∅ meaning positive and $FF meaning negative. If the number is zero, the exponent byte is zero. Your machine code program must be arranged so that it can work with the numbers that are stored in this way.

When the machine code has run, then the result of any work on the number is also placed in the 'floating-point accumulator', and is passed back to BASIC. If you called the machine code by using:

```
PRINT USR(24.5)
```

for example, then the *result* of the machine code calculation will be printed. If you used A=USR(12345), then A contains the result. For many applications of machine code that do not involve the use of floating-point numbers, CALL is a much easier method of running machine code, free from the complications of USR. When you are able to write machine code programs that make use of floating-point numbers, the complications of USR will be much less formidable!

Appendix F
A CTRL Z Reset Action

The Oric-1 and Atmos both use a RESET system which requires a button underneath the computer to be pressed. This button is not very accessible, and in my book on the Oric-1, I suggested gluing on a plastic extension button, using 'superglue'. An alternative is to have a 'soft-action' reset. This is a program which can be loaded in when you first switch on, and which will then allow you to carry out the RESET action by pressing ordinary keys. You can make the program recording in machine code, autorunning, so that it is quick and easy to use. I have tested this only on the Atmos, however.

The principle is that you can reset the Atmos machine by calling the address $F8B2 – the Oric-1 will require a different address, possibly $F430. The cunning part is to make this happen when a set of keys is depressed. As it happens, using the CTRL key along with a letter key will generate a unique code, and the code that is generated when CTRL and Z are pressed at the same time is $1A. If we interrupt the normal keyboard routine with a little piece of code that checks for $1A in the accumulator, we can carry out the 'soft reset'. If $1A is present, the program will jump to the reset address. If this byte is not present, the program returns to normal by jumping to $EB78 (Atmos).

```
   10 A=#400
   20 FORN=0TO20:READ D$
   30 POKEA+N,VAL("#"+D$):NEXT
   40 CALLA
   100 DATAA9,B,8D,3C,2,A9,4,8D,3D,2,60,
 C9,1A,D0,3,20,B2,F8,4C,78,EB
```

Fig. F.1

Since you are by now no longer a novice machine code programmer, I'll simply show the BASIC POKE program (Fig. F1) for this routine, and leave you to work out what has been done, now that you know the principles. The program has been assembled in the memory at $ø4øø.

Index

! command, 116

6502, 7

A register, 41
absolute addressing, 44, 60, 139
accumulator, 41
accumulator actions, 53
address bus, 40
address of VLT, 16
addressing methods, 41
altering memory, 107
ampersand sign (&), 87
AND action, 9
arithmetic and logic, 53
arithmetic set, 8
ASCII codes, 4
assembler, 29, 30, 110
assembly language, 30, 42
attribute, 39
autostart, 123

base address, 46, 91
BEQ, 56, 99
binary code, 2
binary digit, 1
bit, 1
block diagram, 6
BNE, 56
borrow, 51
BRANCH commands, 49
branch displacements, 102
branch instructions, 57
breakpoint, 103
bug, 101
byte, 3

CALL, 58
carry bit, 53
CHRS, 5
clock, 28
clock-pulse generator, 28
CMP, 55, 84
compiler, 27
compiling, 27
complementing, 35
converting hex/binary, 33
counter variable, 73
counting loop, 73
CPU, 7
crash, 59

data bus, 40
data byte, 29
data pins, 10
debugging, 101
decision, 66
decision step, 69
declaring a variable, 15
DEEK, 58
defensive programming, 91
denary scale, 30
denary to hex conversion, 131
destination address, 72
DEX, 47
DEY, 48
disassembly, 109
displacement byte, 71
displacement number, 56
DOKE, 58
dynamically allocated, 17

endless loop, 59
entering text, 107
execution, 111
exponent, 129

faulty loop, 104
filling the screen, 90
flag register, 50

floating point numbers, 129
flowchart shapes, 67
flowcharts, 66

garbage, 14
gates, 28, 40
GTORKB routine, 81

hashmark (#), 33
hex, 30
hex arithmetic, 108
hex scale, 32
hex to denary conversion, 131
HEXS, 33
hexadecimal code, 30
HIMEM, 58
holding loop, 79

ILLEGAL QUANTITY ERROR, 17
immediate addressing, 43, 139
implementing flowchart, 70
implied addressing, 139
incrementing counter, 70
index registers, 86
indexed addressing, 46, 139
indexed indirect, 89
indirect addressing, 48, 89, 139
indirect indexed, 89
initial contents, 102
initialisation routine, 14
input/output, 66
instruction byte, 29
instruction set, 34, 133
integers, 22
interpreted BASIC, 27
INX, 47
INY, 47

JMP, 56
JSR, 81
jump set, 10
junction box, 119

keyboard, 8
keywords, 5

least significant, 3
library of subroutines, 94
line numbers, 25
linear program, 68
loading, 7
logic set, 9
looping program, 68

machine code, 5
mantissa, 129
mantissa-exponent form, 129
memory, 1
memory blocks, 108
message routine, 98
microprocessor, 28
microprocessor unit, 7
mnemonics, 42
monitor actions, 104
most significant, 3
MPU, 7
multiply-by-two, 63

negative numbers, 35
nested loops, 75
NEW action, 72
next-line address, 25

OLD, 113
operand, 42
operator, 42
OR action, 9
ORG, 60
Oricmon, 105
Oricmon assembler, 101

P-register, 50
PC, 40
PEEK, 5
planning, 66
POKE, 17
port, 12, 100
practical programs, 60
precision, 23
prevention of faults, 101
problem sorting, 111
process, 66
Processor Status register, 50
program counter, 40
program protection, 122
program storage, 24
PULL, 59
pulling, 118
PUSH, 59
pushing, 118

RAM, 4
random access memory, 4
reading, 10
real numbers, 22
registers, 40
relative addressing, 49

relocatable program, 127
REM, 60
renumbering routine, 124
repeated character printing, 83
reset bit, 51
RESTORE button, 18
return from subroutine, 59
ROM, 3
ROM subroutines, 80
ROTATE, 53
rounding, 23
RTS, 59
running a program, 26

S-register, 49
saving on tape, 95
screen, 8
screen displays, 37
SEC, 77
set bit, 50
SHIFT, 53
shifting loop, 68
shoot utility, 119
signed number, 35
significance, 3
source address, 72
source code, 110
stack, 59, 117
stack pointer, 49
staging post, 83
starting byte, 58
states, 2
STOP disable, 123
storing, 7
storing a program, 24
string storage, 19
subroutines, 14
SUBSET series, 103
syntax error, 26
system use, 15

terminator, 66, 97
test and branch, 56
text screen memory, 37
text window, 38
time delay, 73
token byte, 4
tracing, 112
transfer of byte, 53
types of memory, 3

unit of memory, 1
unsigned number, 35
USR command, 63
utilities, 113

variable list table (VLT), 15
VDU routine, 81
video loops, 86
VLT, 15
VLT address, 16
VLT for integers, 22
VLT for number, 18
VLT for string, 19

X-indirect, 89
XOR action, 9

Y-indirect, 89

zero page addressing, 45, 79, 139
zero-page memory, 89

Sooner or later most users feel the need to use machine code to achieve higher speeds, more spectacular effects and better control. This book is an essential introduction to machine code for Oric and Atmos owners who take their computing seriously and want to get higher performance from their machine.

Many illustrative programs are included to help you get started, and everything is set out in a practical way with explanations that are easy to understand. You will gain valuable insight into the working of the machine, and the book gives the most useful entry points and memory locations. Soon you will be amazing yourself with your own versatility!

The Author
Ian Sinclair is a well known and regular contributor to journals such as *Personal Computer World, Computing Today, Electronics and Computing Monthly, Hobby Electronics* and *Electronics Today International*. He has written over forty books on aspects of electronics and computing, mainly aimed at the beginner.

Front cover illustration by Alan Craddock

GRANADA PUBLISHING
Printed in Great Britain 0 246 12150 5

£6.95 net